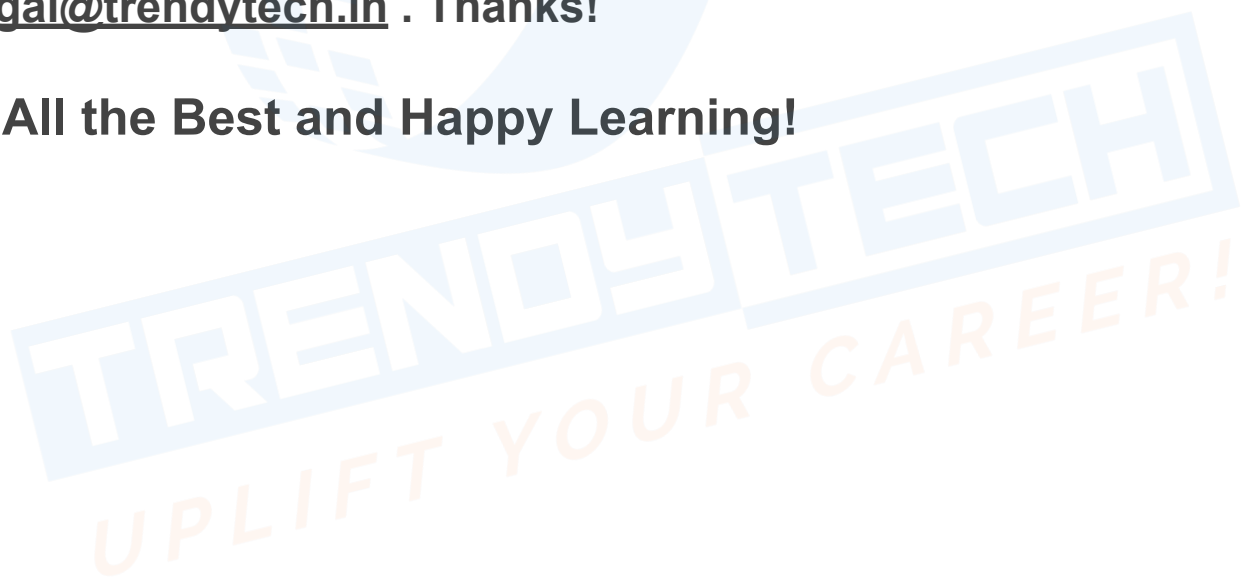


Disclaimer: These slides are copyrighted and strictly for personal use only

- This document is reserved for people enrolled into the [Ultimate Big Data Masters Program \(Cloud Focused\) by Sumit Sir](#)
- **Please do not share this document**, it is intended for personal use and exam preparation only, thank you.
- If you've obtained these slides for free on a website that is not the course's website, please reach out to legal@trendytech.in . Thanks!
- All the Best and Happy Learning!



Apache PySpark Project

Loan score calculation is dependent on 3 factors -

1. Loan Payment History (Loan repayments history for previous loans if any)
2. Customer's Financial Health
3. Loan Defaulters History (Delinq, Public records, Bankruptcies, Enquiries)

In order to calculate the loan score, two important tables are required

- Delinq
- A table with columns consisting of the details of Public records, Bankruptcies, Enquiries

Cleaning the records and creating a Processed Dataframe

```
loans_def_processed_df = loans_def_raw_df.withColumn("delinq_2yrs",  
col("delinq_2yrs").cast("integer").fillna(0, subset=["delinq_2yrs"]))
```

```
loans_def_processed_pub_rec_df = loans_def_raw_df.withColumn("pub_rec",  
col("pub_rec").cast("integer").fillna(0, subset=["pub_rec"]))
```

```
loans_def_processed_pub_rec_bankruptcies_df =  
loans_def_processed_pub_rec_df.withColumn("pub_rec_bankruptcies",  
col("pub_rec_bankruptcies").cast("integer").fillna(0,  
subset=["pub_rec_bankruptcies"]))
```

```
loans_def_processed_inq_last_6mths_df =  
loans_def_processed_pub_rec_bankruptcies_df.withColumn("inq_last_6mths",  
col("inq_last_6mths").cast("integer").fillna(0, subset=["inq_last_6mths"]))
```

Creating a temporary table on the above processed data

```
loans_def_processed_inq_last_6mths_df.  
createOrReplaceTempView("loan_defaulters")
```

Creating a detailed Dataframe including the above mentioned columns from the loan_defaulters data used for the calculation of loan score.

```
loan_defaulters_detail_records_enq_df = spark.sql("select member_id, pub_rec, pub_rec_bankruptcies, inq_last_6mths from loan_defaulters")
```

Writing back the Processed data to the Cleaned folder

Writing back in CSV Format

```
loan_defaulters_detail_records_enq_df.write \  
.option("header", True) \  
.format("csv") \  
.mode("overwrite") \  
.option("path", \  
"/public/trendytech/lendingclubproject/cleaned/Loan_defaulters_detail_records_enq_csv") \  
.save()
```

Writing back in PARQUET Format

```
loan_defaulters_detail_records_enq_df.write \  
.format("parquet") \  
.mode("overwrite") \  
.option("path", \  
"/public/trendytech/lendingclubproject/cleaned/Loan_defaulters_detail_records_enq_csv") \  
.save()
```

Final Cleaned and Processed Datasets for future processing -

Customers

```
member_id
emp_title
emp_length
home_ownership
annual_income
address_state
address_zipcode
address_country
grade
sub_grade
verification_status
tot_hi_cred_lim
application_type
join_annual_income
verification_status_joint
ingest_date
```

Loans

```
loan_id
member_id
loan_amount
funded_amount
loan_term_months
interest_rate
monthly_installment
issue_date
loan_status
loan_purpose
loan_title
ingest_date
```

Loan Repayments

```
loan_id
total_principal_received
total_interest_received
total_late_fee_received
total_payment_received
last_payment_amount
last_payment_date
next_payment_date
ingest_date
```

Loan Defaulters

```
member_id
delinq_2yrs
delinq_amnt
pub_rec
pub_rec_bankruptcies
inq_last_6mths
total_rec_late_fee
mths_since_last_delinq
mths_since_last_record
```

Loan Defaulters Delinq

```
member_id
delinq_2yrs
delinq_amnt
mths_since_last_delinq
```

Loan Defaulters Detail

```
member_id
pub_rec
pub_rec_bankruptcies
inq_last_6mths
```

Permanent Table Creation on the Cleaned Data -

Business Requirement 1 : Some of the teams are required to analyse the cleaned data which requires the creation of permanent tables on top of the cleaned data that allows the downstream teams to query the data using simple SQL like queries.

Note :

- Table comprises of Actual Data & Metadata(schema)
- What kind of table needs to be created in this scenario as multiple teams are accessing the data?

There are 2 kinds of tables.

1. **Managed Table** : Since the data and metadata are placed in a specified default location, on dropping a Managed table, both the Data and the Metadata will be lost.

2. **External Table** : Since the data is placed in an external location, dropping an external table will only delete the Metadata/Schema but the actual data is not lost.

- For the above use-case, since multiple teams are accessing the data, it is a best practice to create external tables as it doesn't affect the actual data even if the table is dropped accidentally.

In the case of Managed Tables, Data is stored in the warehouse directory as mentioned in the configuration options while creating the spark session. Spark uses hive metastore to store the metadata in a persistent way.

```
from pyspark.sql import SparkSession
import getpass
username=getpass.getuser()
spark=SparkSession. \
    builder. \
    config('spark.ui.port','0'). \
    config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
    config('spark.shuffle.useOldFetchProtocol', 'true'). \
    enableHiveSupport(). \
    master('yarn'). \
    getOrCreate()
```

Reading the cleaned data and creating Dataframe for further processing

```
customers_df = spark.read \
    .format("parquet") \
    .load("/public/trendytech/lendingclubproject/cleaned/customers_parquet")
```

Steps for creating a Permanent Table

1. Create a Database with a proper naming convention to recognize the data easily.

```
spark.sql("create database itv005857_lending_club")
```

2. Create an External Table by providing the fully qualified table name (database_name.table_name)

```
spark.sql("""create external table itv005857_lending_club.customers(
member_id string, emp_title string, emp_length int,
home_ownership string, annual_income float, address_state string, address_zipcode string, address_country string, grade string,
sub_grade string, verification_status string, total_high_credit_limit float, application_type string, join_annual_income float,
verification_status_joint string, ingest_date timestamp)
stored as parquet location '/public/trendytech/lendingclubproject/cleaned/customers_parquet'
""")
```

3. To view the data

```
spark.sql("select * from itv005857_lending_club.customers")
```

Data can be viewed in the Hive terminal as well.

Command - describe formatted <table-name>

(will give all the details of the table, like - owner of the table, type of the table, and so on)

Business Requirement 2 : The teams require a single consolidated view of all the datasets with the latest up-to-date data.

Solution : The best practice would be to create a view on the cleaned data that refreshes every 24 hrs. So the data that is part of the view will not be older than 24 hrs.

Creating a View - create or replace view <view-name> as select-query



```
spark.sql("""
create or replace view itv005857_lending_club.customers_loan_v as select
l.loan_id,
c.member_id,
c.emp_title,
c.emp_length,
c.home_ownership,
c.annual_income,
c.address_state,
c.address_zipcode,
c.address_country,
c.grade,
c.sub_grade,
c.verification_status,
c.total_high_credit_limit,
c.application_type,
c.join_annual_income,
c.verification_status_joint,
l.loan_amount,
l.funded_amount,
l.loan_term_years,
l.interest_rate,
l.monthly_installment,
l.issue_date,
l.loan_status,
l.loan_purpose,
r.total_principal_received,
r.total_interest_received,
r.total_late_fee_received,
r.last_payment_date,
r.next_payment_date,
d.delinq_2yrs,
d.delinq_amnt,
d.mths_since_last_delinq,
e.pub_rec,
e.pub_rec_bankruptcies,
e.inq_last_6mths

FROM itv006277_lending_club.customers c
LEFT JOIN itv006277_lending_club.loans l on c.member_id = l.member_id
LEFT JOIN itv006277_lending_club.loans_repayments r ON l.loan_id = r.loan_id
LEFT JOIN itv006277_lending_club.loans_defaulters_delinq d ON c.member_id = d.member_id
LEFT JOIN itv006277_lending_club.loans_defaulters_detail_rec_enq e ON c.member_id = e.member_id
""")
```

Note: Creating a view would be much faster as there is no actual data processing taking place. However, a query to view the data, like the following -

```
spark.sql("select * from itv005857_lending_club.customers_loan_v")
```

This query will take time to execute as it involves joining multiple tables to generate a view with the desired data.

Business Requirement 2 : Yet another team wants real quick access to the “view data” without having to wait for the view results to be processed. Since processing the results takes a very long time.

Solution : Precalculate the results by executing the join of tables prior. Ex - A weekly job performs the join of the underlying tables and stores the results in another table.

Disadvantage : Even though the results are pre-calculated and can be fetched much faster than the previous case, the data is not the latest up-to-date data but rather a week old data. This approach can be taken if it is okay to consider a little older data for further processing.

Note: In this case, a Managed Table is created. The actual data for this table will be stored in the warehouse directory and the metadata is present in the Hive metastore.

Loan Score Calculation Criterias

Higher the Loan Score, better the chances of getting the loan approval and vice-versa.

3 major factors affecting the Loan Score :

1. Loan Repayment History

This factor based on - last_payment & total_payment_received

2. Loan Defaulters History

This factor is based on - delinq_2years, public_rec, public_rec_bankruptcies & inq_last_6mths

3. Financial Health

This factor is based on - home_ownership, loan_status, funded_amount, grade_points

% contribution of each of the factors for the loan score calculation -

Loan Repayment History - 20%

Loan Defaulters History - 45%

Financial Health - 35%

Identifying the Bad Data and Final Cleaning (Repeating Member IDs)

The repeating member ids would be bad data as there are multiple records for a single member-id (Ideally, there should be one record associated with a member-id).

- Query to identify the bad data :

```
bad_data_customer_df = spark.sql("""select member_id from(select member_id, count(*)  
as total from itv006277_lending_club.customers  
group by member_id having total > 1)""")
```

Displays all the member-ids which have more than 1 records associated, implying the bad data.

- Create a consolidated CSV file which consists of the bad data from all the dataframes (Union of all the dataframes and choose the unique member-ids). This file can then be shared with the upstream team for correction of the bad data.

```
bad_data_loans_defaulters_detail_rec_enq_df.repartition(1).write \  
.format("csv") \  
.option("header", True) \  
.mode("overwrite") \  
.option("path", "/user/itv006277/lendingclubproject/bad/bad_data_loans_defaulters_detail_rec_enq") \  
.save()
```


- Create a temporary view of the bad data file.
- Segregate the Good Data and create a final cleaned dataframe by excluding the member-ids present in the bad data temporary view. Store the final cleaned data to a new_cleaned folder

```
bad_customer_data_final_df.createOrReplaceTempView("bad_data_customer")
```

```
customers_df = spark.sql("""select * from itv006277_lending_club.customers
where member_id NOT IN (select member_id from bad_data_customer)
""")
```

```
customers_df.write \
    .format("parquet") \
    .mode("overwrite") \
    .option("path", "/user/itv006277/lendingclubproject/raw/cleaned_new/customers_parquet") \
    .save()
```

- Create External Tables over the final new cleaned data.

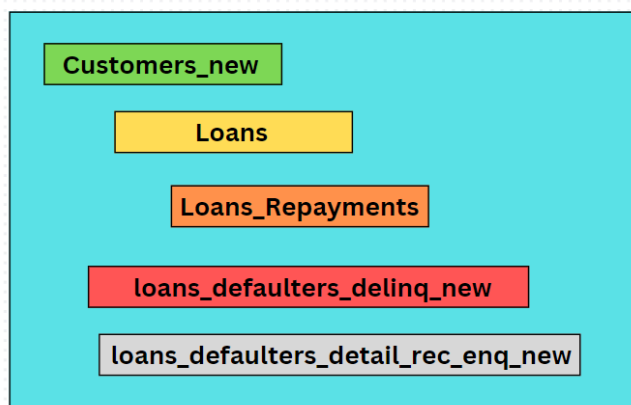
```
spark.sql("""
create EXTERNAL TABLE itv006277_lending_club.customers_new(member_id string, emp_title string, emp_length int, home_ownership string,
annual_income float, address_state string, address_zipcode string, address_country string, grade string,
sub_grade string, verification_status string, total_high_credit_limit float, application_type string,
join_annual_income float, verification_status_joint string, ingest_date timestamp)
stored as parquet
LOCATION '/public/trendytech/lendingclubproject/cleaned_new/customer_parquet'
""")
```

Note: Repeat the same for the other datasets

- Create Spark Session
- Need to configure user defined variables as shown in the diagram below (Code modifications are easier when the variable values are defined separately as compared to hardcoded values.)

```
spark.conf.set("spark.sql.unacceptableRatedPts", 0)
spark.conf.set("spark.sql.veryBadRatedPts", 100)
spark.conf.set("spark.sql.badRatedPts", 250)
spark.conf.set("spark.sql.goodRatedPts", 500)
spark.conf.set("spark.sql.veryGoodRatedPts", 650)
spark.conf.set("spark.sql.excellentRatedPts", 800)
```

Tables required to calculate the Loan Score



- Factors that contribute to the Loan score calculation

Loan Repayment History - 20%

```

ph_df = spark.sql("select c.member_id, \
case \
when p.last_payment_amount < (c.monthly_installment * 0.5) then ${spark.sql.very_badRated_pts} \
when p.last_payment_amount >= (c.monthly_installment * 0.5) and p.last_payment_amount < c.monthly_installment then ${spark.sql.very_badRated_pts} \
when (p.last_payment_amount = (c.monthly_installment)) then ${spark.sql.goodRated_pts} \
when p.last_payment_amount > (c.monthly_installment) and p.last_payment_amount <= (c.monthly_installment * 1.50) then ${spark.sql.very_goodRated_pts} \
when p.last_payment_amount > (c.monthly_installment * 1.50) then ${spark.sql.excellentRated_pts} \
else ${spark.sql.unacceptableRated_pts} \
end as last_payment_pts, \
case \
when p.total_payment_received >= (c.funded_amount * 0.50) then ${spark.sql.very_goodRated_pts} \
when p.total_payment_received < (c.funded_amount * 0.50) and p.total_payment_received > 0 then ${spark.sql.goodRated_pts} \
when p.total_payment_received = 0 or (p.total_payment_received) is null then ${spark.sql.unacceptableRated_pts} \
end as total_payment_pts \
from itv005857_lending_club.loans_repayments p \
inner join itv005857_lending_club.loans c on c.loan_id = p.loan_id where member_id NOT IN (select member_id from bad_data_customer)")
  
```

Loan Defaulters History - 45%

```

ldh_ph_df = spark.sql(
"select p.*, \
CASE \
WHEN d.delinq_2yrs = 0 THEN ${spark.sql.excellentRated_pts} \
WHEN d.delinq_2yrs BETWEEN 1 AND 2 THEN ${spark.sql.badRated_pts} \
WHEN d.delinq_2yrs BETWEEN 3 AND 5 THEN ${spark.sql.very_badRated_pts} \
WHEN d.delinq_2yrs > 5 OR d.delinq_2yrs IS NULL THEN ${spark.sql.unacceptableGrade_pts} \
END AS delinq_pts, \
CASE \
WHEN l.pub_rec = 0 THEN ${spark.sql.excellentRated_pts} \
WHEN l.pub_rec BETWEEN 1 AND 2 THEN ${spark.sql.badRated_pts} \
WHEN l.pub_rec BETWEEN 3 AND 5 THEN ${spark.sql.very_badRated_pts} \
WHEN l.pub_rec > 5 OR l.pub_rec IS NULL THEN ${spark.sql.very_badRated_pts} \
END AS public_records_pts, \
CASE \
WHEN l.pub_rec_bankruptcies = 0 THEN ${spark.sql.excellentRated_pts} \
WHEN l.pub_rec_bankruptcies BETWEEN 1 AND 2 THEN ${spark.sql.badRated_pts} \
WHEN l.pub_rec_bankruptcies BETWEEN 3 AND 5 THEN ${spark.sql.very_badRated_pts} \
WHEN l.pub_rec_bankruptcies > 5 OR l.pub_rec_bankruptcies IS NULL THEN ${spark.sql.very_badRated_pts} \
END as public_bankruptcies_pts, \
CASE \
WHEN l.inq_last_6mths = 0 THEN ${spark.sql.excellentRated_pts} \
WHEN l.inq_last_6mths BETWEEN 1 AND 2 THEN ${spark.sql.badRated_pts} \
WHEN l.inq_last_6mths BETWEEN 3 AND 5 THEN ${spark.sql.very_badRated_pts} \
WHEN l.inq_last_6mths > 5 OR l.inq_last_6mths IS NULL THEN ${spark.sql.unacceptableRated_pts} \
END AS enq_pts \
FROM itv005857_lending_club.loans_defaulters_detail_rec_enq_new l \
INNER JOIN itv005857_lending_club.loans_defaulters_delinq_new d ON d.member_id = l.member_id \
INNER JOIN ph_pts p ON p.member_id = l.member_id where l.member_id NOT IN (select member_id from bad_data_customer)")
  
```

Financial Health - 35%

```
fh_ldh_ph_df = spark.sql("select ldef.*, \nCASE \nWHEN LOWER(l.loan_status) LIKE '%Fully paid%' THEN ${spark.sql.excellent rated pts} \nWHEN LOWER(l.loan_status) LIKE '%current%' THEN ${spark.sql.good rated pts} \nWHEN LOWER(l.loan_status) LIKE '%in grace period%' THEN ${spark.sql.bad rated pts} \nWHEN LOWER(l.loan_status) LIKE '%late (16-30 days)%' OR LOWER(l.loan_status) LIKE '%late (31-120 days)%' THEN ${spark.sql.very_bad rated pts} \nWHEN LOWER(l.loan_status) LIKE '%charged off%' THEN ${spark.sql.unacceptable rated pts} \nelse ${spark.sql.unacceptable rated pts} \nEND AS loan_status_pts, \nCASE \nWHEN LOWER(a.home_ownership) LIKE '%own' THEN ${spark.sql.excellent rated pts} \nWHEN LOWER(a.home_ownership) LIKE '%rent' THEN ${spark.sql.good rated pts} \nWHEN LOWER(a.home_ownership) LIKE '%mortgage' THEN ${spark.sql.bad rated pts} \nWHEN LOWER(a.home_ownership) LIKE '%any' OR LOWER(a.home_ownership) IS NULL THEN ${spark.sql.very_bad rated pts} \nEND AS home_pts, \nCASE \nWHEN l.funded_amount <= (a.total_high_credit_limit * 0.10) THEN ${spark.sql.excellent rated pts} \nWHEN l.funded_amount > (a.total_high_credit_limit * 0.10) AND l.funded_amount <= (a.total_high_credit_limit * 0.20) THEN ${spark.sql.very_good rated pts} \nWHEN l.funded_amount > (a.total_high_credit_limit * 0.20) AND l.funded_amount <= (a.total_high_credit_limit * 0.30) THEN ${spark.sql.good rated pts} \nWHEN l.funded_amount > (a.total_high_credit_limit * 0.30) AND l.funded_amount <= (a.total_high_credit_limit * 0.50) THEN ${spark.sql.bad rated pts} \nWHEN l.funded_amount > (a.total_high_credit_limit * 0.50) AND l.funded_amount <= (a.total_high_credit_limit * 0.70) THEN ${spark.sql.very_bad rated pts} \nelse ${spark.sql.unacceptable rated pts} \nEND AS credit_limit_pts, \nCASE \nWHEN (a.grade) = 'A' and (a.sub_grade)='A1' THEN ${spark.sql.excellent rated pts} \nWHEN (a.grade) = 'A' and (a.sub_grade)='A2' THEN (${spark.sql.excellent rated pts} * 0.95) \nWHEN (a.grade) = 'A' and (a.sub_grade)='A3' THEN (${spark.sql.excellent rated pts} * 0.90) \nWHEN (a.grade) = 'A' and (a.sub_grade)='A4' THEN (${spark.sql.excellent rated pts} * 0.85) \nWHEN (a.grade) = 'A' and (a.sub_grade)='A5' THEN (${spark.sql.excellent rated pts} * 0.80) \nWHEN (a.grade) = 'B' and (a.sub_grade)='B1' THEN ${spark.sql.very_good rated pts} \nWHEN (a.grade) = 'B' and (a.sub_grade)='B2' THEN (${spark.sql.very_good rated pts} * 0.95) \nWHEN (a.grade) = 'B' and (a.sub_grade)='B3' THEN (${spark.sql.very_good rated pts} * 0.90) \nWHEN (a.grade) = 'B' and (a.sub_grade)='B4' THEN (${spark.sql.very_good rated pts} * 0.85) \nWHEN (a.grade) = 'B' and (a.sub_grade)='B5' THEN (${spark.sql.very_good rated pts} * 0.80) \nWHEN (a.grade) = 'C' and (a.sub_grade)='C1' THEN ${spark.sql.good rated pts} \nWHEN (a.grade) = 'C' and (a.sub_grade)='C2' THEN (${spark.sql.good rated pts} * 0.95) \nWHEN (a.grade) = 'C' and (a.sub_grade)='C3' THEN (${spark.sql.good rated pts} * 0.90) \nWHEN (a.grade) = 'C' and (a.sub_grade)='C4' THEN (${spark.sql.good rated pts} * 0.85) \nWHEN (a.grade) = 'C' and (a.sub_grade)='C5' THEN (${spark.sql.good rated pts} * 0.80) \nWHEN (a.grade) = 'D' and (a.sub_grade)='D1' THEN ${spark.sql.bad rated pts} \nWHEN (a.grade) = 'D' and (a.sub_grade)='D2' THEN (${spark.sql.bad rated pts} * 0.95) \nWHEN (a.grade) = 'D' and (a.sub_grade)='D3' THEN (${spark.sql.bad rated pts} * 0.90) \nWHEN (a.grade) = 'D' and (a.sub_grade)='D4' THEN (${spark.sql.bad rated pts} * 0.85) \nWHEN (a.grade) = 'D' and (a.sub_grade)='D5' THEN (${spark.sql.bad rated pts} * 0.80) \nWHEN (a.grade) = 'E' and (a.sub_grade)='E1' THEN ${spark.sql.very_bad rated pts} \nWHEN (a.grade) = 'E' and (a.sub_grade)='E2' THEN (${spark.sql.very_bad rated pts} * 0.95) \nWHEN (a.grade) = 'E' and (a.sub_grade)='E3' THEN (${spark.sql.very_bad rated pts} * 0.90) \nWHEN (a.grade) = 'E' and (a.sub_grade)='E4' THEN (${spark.sql.very_bad rated pts} * 0.85) \nWHEN (a.grade) = 'E' and (a.sub_grade)='E5' THEN (${spark.sql.very_bad rated pts} * 0.80) \nWHEN (a.grade) in ('F', 'G') THEN (${spark.sql.unacceptable rated pts}) \nEND AS grade_pts \nFROM ldh_ph_pts ldef \nINNER JOIN itv005857_lending_club_loans l ON ldef.member_id = l.member_id \nINNER JOIN itv005857_lending_club_customers_new a ON a.member_id = ldef.member_id where ldef.member_id NOT IN (select member_id from bad_data_customer)")
```

- Final Loan Score Calculation :

Final Loan score results will be stored under the processed folder in HDFS.

```
loan_score = spark.sql("SELECT member_id, \n((last_payment_pts+total_payment_pts)*0.20) as payment_history_pts, \n((delinq_pts + public_records_pts + public_bankruptcies_pts + enq_pts) * 0.45) as defaulters_history_pts, \n((loan_status_pts + home_pts + credit_limit_pts + grade_pts)*0.35) as financial_health_pts \nFROM fh_ldh_ph_pts")\n\nfinal_loan_score = loan_score.withColumn('loan_score',loan_score.payment_history_pts + loan_score.defaulters_history_pts + loan_score.financial_health_pts)\n\nfinal_loan_score.createOrReplaceTempView("loan_score_eval")\n\nloan_score_final = spark.sql("select ls.*, \ncase \nWHEN loan_score > ${spark.sql.very_good_grade_pts} THEN 'A' \nWHEN loan_score <= ${spark.sql.very_good_grade_pts} AND loan_score > ${spark.sql.good_grade_pts} THEN 'B' \nWHEN loan_score <= ${spark.sql.good_grade_pts} AND loan_score > ${spark.sql.bad_grade_pts} THEN 'C' \nWHEN loan_score <= ${spark.sql.bad_grade_pts} AND loan_score > ${spark.sql.very_bad_grade_pts} THEN 'D' \nWHEN loan_score <= ${spark.sql.very_bad_grade_pts} AND loan_score > ${spark.sql.unacceptable_grade_pts} THEN 'E' \nWHEN loan_score <= ${spark.sql.unacceptable_grade_pts} THEN 'F' \nend as loan_final_grade \nfrom loan_score_eval ls")
```

Project Structuring

Notebooks are a best means of exploring and developing the required logic to meet the business requirements.

However, in real-time, IDEs like Pycharm / Visual Studio code will be required to check for the compatibility of the new code with the existing project, package and deploy the code.

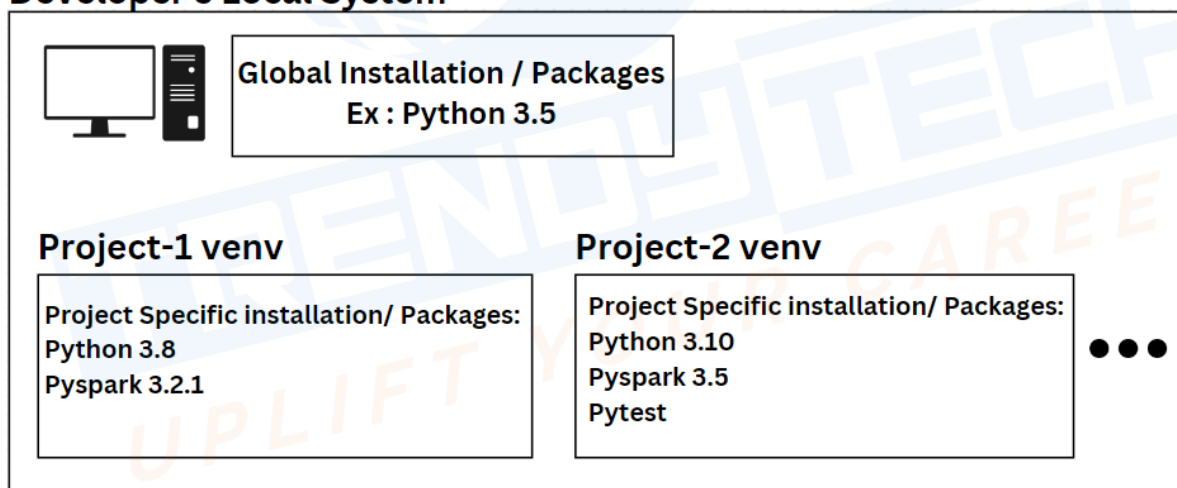
Approach to work on a project and the essential setup required for the project

- As a developer you would be working on your local repository of the project and then push the newly developed & tested code to the central repository.
- pipenv is a utility used to install the required project dependencies/packages and also creating a virtual environment for the project.

pipenv = pip + venv

venv (virtual environment) is used to create an isolated virtual environment for a project consisting of project specific installations and package versions.

Developer's Local System



Notebooks are primarily used for exploration purposes, but there are occasions where using an integrated development environment (IDE) like VS Code or PyCharm becomes necessary.

To set up a virtual environment for your project, follow these steps:

1. **Install Pipenv:** Begin by installing Pipenv using the command “pip install pipenv”.
2. **Enter the Virtual Environment:** Use the command “pipenv shell” to enter the virtual environment.
3. **Install Packages:** Install required packages directly into the virtual environment.

For example: To install pyspark, use “pipenv install pyspark”

4. **Uninstall Packages:** To remove packages from the virtual environment, utilize command “pipenv uninstall <package>”.

Note: Package versions are managed in the `pipfile.lock`.

5. **Remove the Virtual Environment:** If needed, remove the virtual environment entirely with “pipenv –rm”.

For installing a specific package version, such as pyspark 3.2.1:

- a. **Remove Existing Environment:** Execute “pipenv –rm” to remove the existing virtual environment.
- b. **Specify Package Version:** Update the `pipfile` to include the desired package version, e.g., `pyspark = "==3.2.1"`.
- c. **Recreate the Virtual Environment:** Recreate the virtual environment by running the command “pipenv install”.
- d. **Enter the New Environment:** Access the updated virtual environment using “pipenv shell”.

Through these steps, virtual environments facilitate the management of package versions tailored to specific project requirements, ensuring project isolation and dependency control.

Pipfile & Pipfile.lock

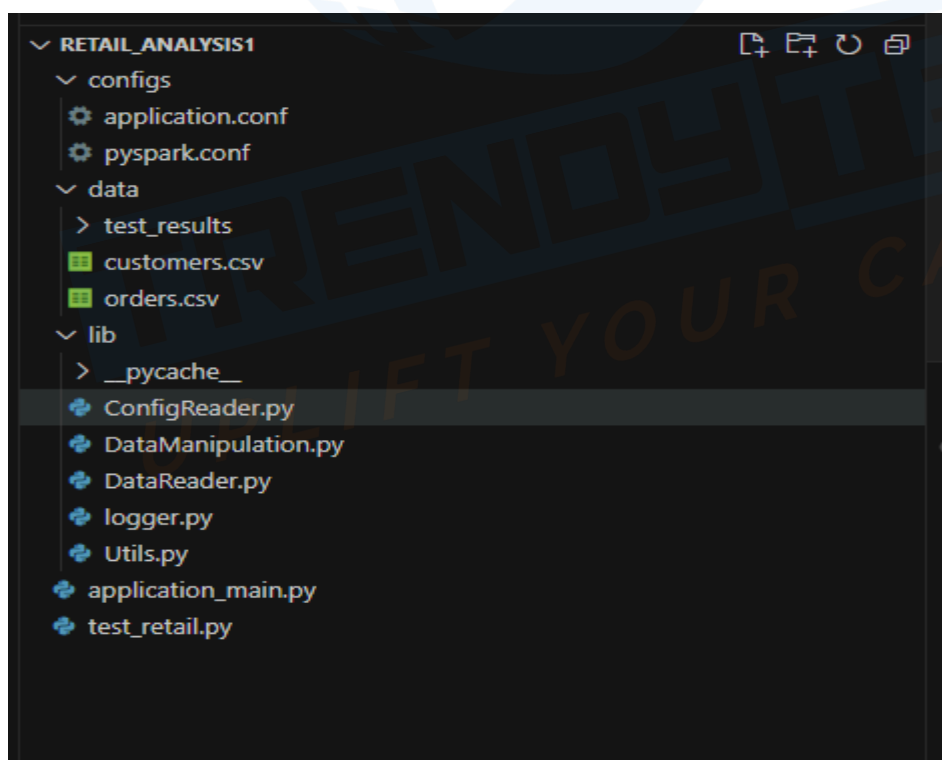
- A Pipfile is a configuration file used by Python projects that utilize the pipenv tool for managing dependencies and virtual environments. It defines the project's dependencies and specifies the Python version required for the project.
- On the other hand, a Pipfile.lock file is generated by pipenv when you install dependencies for your project. It's a lock file that records the exact versions of all dependencies and their transitive dependencies, along with their hashes.

Pyenv

Pyenv is a tool used for managing multiple Python installations on a single system. It allows you to easily switch between different versions of Python, set the global Python version, and manage Python versions on a per-project basis.

Problem Statement: To find the number of closed orders for each state.

Step 1: Create the retail_analysis project having below files.



Step 2: Write the code to find the solution for the problem statement as shown below screenshot. And run this application.py with the environment “LOCAL”

```
application_main.py
1 import sys
2 from lib import DataManipulation, DataReader, Utils, logger
3 from pyspark.sql.functions import *
4 from lib.logger import Log4j
5
6 if __name__ == '__main__':
7     if len(sys.argv) < 2:
8         print("Please specify the environment")
9         sys.exit(-1)
10
11     job_run_env = sys.argv[1]
12     print("Creating SparkSession")
13
14     spark = Utils.get_spark_session(job_run_env)
15     print("Created SparkSession")
16     logger = Log4j(spark)
17     logger.warn("Created Spark Session")
18
19     orders_df = DataReader.read_orders(spark, job_run_env)
20
21     orders_filtered = DataManipulation.filter_closed_orders(orders_df)
22
23     customers_df = DataReader.read_customers(spark, job_run_env)
24
25     joined_df = DataManipulation.join_orders_customers(orders_filtered, customers_df)
26
27     aggregated_results = DataManipulation.count_orders_state(joined_df)
28
29     aggregated_results.show()
30
31     print("end of main")
```


Unit testing

- If we have written our code in a modular way, then we can test each function separately, ensuring that each unit of code behaves as expected in isolation. This approach, known as unit testing
- The framework used for unit testing are unittest, pytest. And in our course we will basically use the pytest framework.
- To install pytest run the code “pipenv install pytest” and its exact version will be visible in Pipfile.lock.
- The unit test cases typically start with the prefix "test_". This naming convention helps Pytest automatically discover and execute the test cases when you run your test suite. Also the filename where you write your unit test cases should either start or end with “test”.

Using Unit test cases we will check below conditions:

1. read_customers_df - 12435
2. read_orders_df - 68883f
3. filter_closed_orders - 7556
4. read_app_config (Using read_app_config we will verify all the configurations.)

The unit test code is shown in the below screenshot.



```
test_retail.py
test_retail.py > ...
1
2 import pytest
3 from lib.DataReader import read_customers, read_orders
4 from lib.DataManipulation import filter_closed_orders, count_orders_state, filter_orders_generic
5 from lib.ConfigReader import get_app_config
6
7
8 def test_customerdf_count(spark):
9     customer_count = read_customers(spark, "LOCAL").count()
10    assert customer_count == 12435
11
12 def test_orders_data(spark):
13     orders_count = read_orders(spark, "LOCAL").count()
14     assert orders_count == 68883
15
16 def test_filter_closed_orders(spark):
17     orders_df = read_orders(spark, "LOCAL")
18     closed_orders = filter_closed_orders(orders_df).count()
19     assert closed_orders == 7556
20
21 def test_read_app_config():
22     config = get_app_config("LOCAL")
23     assert config["orders.file.path"] == "data/orders.csv"
```

To run the test run the code:

“python -m pytest”.

As the python file is present in the virtual environment complete path of python.exe file we will mention so the command will become

Ex:

C:\Users\HP\.virtualenvs\Retail_Analysis_2-GE1CUWNa\Scripts\python.exe
-m pytest

Once the test cases are passed it will be mentioned in error logs

To get extra information/result use “-v” so command will be:

“python -m pytest -v”

Ex:

```
C:\Users\HP\.virtualenvs\Retail_Analysis_2-GE1CUWNa\Scripts\python.exe  
-m pytest -v
```

Fixtures

- Fixtures are defined using the “@pytest.fixture” decorator. And they are especially useful for reusable setup code. Ideally fixtures should be written in a separate file like “conftest.py” and no need to specify where it is written as pytest framework will directly use fixtures present in this file.
- This fixture creates a SparkSession object for testing and returns it to the test functions that request it. It provides a convenient way to set up a Spark environment for testing without having to repeat the setup code in each test.

```
@pytest.fixture  
def spark():  
    spark_session = get_spark_session("LOCAL")  
    return spark_session
```

- Fixtures can optionally include teardown(releasing the resources) code to clean up resources after the test runs. This is done using the “yield statement”. Anything after the yield statement serves as teardown code.
- The yield statement marks the point where the test setup ends and the cleanup begins. The fixture yields the SparkSession object to the test function that uses it. The test function will execute with the provided SparkSession.
- After the test function finishes executing, the fixture performs cleanup by stopping the SparkSession. This releases any resources held by the SparkSession and ensures that it is properly shut down.

```
@pytest.fixture
def spark():
    spark_session = get_spark_session("LOCAL")
    yield spark_session
    spark_session.stop()
```

Command to see list of all the fixtures:
 “python -m pytest --fixtures”

Markers

Markers in Pytest are utilized to label or annotate specific test cases, allowing you to identify and selectively execute those particular test cases during test runs.

In the code snippet below, certain test cases are marked with the "latest" marker. To specifically execute these marked test cases, you can use the following command:

C:\Users\HP\virtualenvs\Retail_Analysis_2-GE1CUWNa\Scripts\python.exe
 -m pytest -m latest

```
@pytest.mark.latest()
def test_check_closed_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count = filter_orders_generic(orders_df,"CLOSED").count()
    assert filtered_count == 7556

@pytest.mark.latest()
def test_check_pendingpayment_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count =
    filter_orders_generic(orders_df,"PENDING_PAYMENT").count()
    assert filtered_count == 15030

@pytest.mark.latest()
def test_check_complete_count(spark):
    orders_df = read_orders(spark,"LOCAL")
    filtered_count = filter_orders_generic(orders_df,"COMPLETE").count()
    assert filtered_count == 22899
```

Command to see list of all the fixtures:
 “python -m pytest --markers”

To reduce redundancy and improve code readability, we can parameterize the test function. This allows us to write a more generic test function that can handle different scenarios by passing parameters. By doing so, we can avoid duplicating code for similar test cases and make our test suite more maintainable.

Logging in Apache Spark

Issues with print statements:

1. We cannot set the logging level like WARN, INFO, ERROR etc
2. If your application contains numerous print statements, you'll face the tedious task of either manually commenting out or removing each one individually.
3. Using print statements in your applications can degrade performance.

So implementing a logging framework such as Log4j provides a structured and efficient approach to managing application logs compared to using print statements.

- Spark internally uses Log4j for its logging, so we can reuse the same for application level logs.
- The logging levels in Apache Spark follow a hierarchical order, starting from the most detailed level, debug, followed by info, warn, error, and finally, fatal.
- In your log4j.properties file if you have defined let's say logging level as "warn" then all logs for that level and level with higher priorities will be visible i.e. warn, error, fatal.
- Using the code you can define the logging level and can mention whether we want to see logs to console or want to save in the file.
log4j.rootCategory=INFO, consol
- To ensure the system incorporates the specified 'log4j.properties' configuration, we will enhance the Spark session creation code in Utils.py by adding the following configuration: `.config('spark.driver.extraJavaOptions', '-Dlog4j.configuration=file:log4j.properties')`.
- Once the Spark session has been configured in the 'application_main.py' file, logging can be enabled by adding the line `'logger = Log4j(spark)'`. With this setup in place, the updated file is ready to be executed.