Customer Value

CLV stands for "Customer Lifetime Value". calcularion using Spark/PySpark and FRM (Frequency, Recency, and Monetary Value) - one method used to segment customers based on their purchase behavior.

In [1]:
```
## Athena for Spark Job already have spark session set-up
# spark
```

In [1]:
```
spark.version
```

```
Calculation started (calculation_id=5ec38d12-53ef-2862-c7ce-db9293a08772) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
'3.2.1-amzn-0'
```

Aux functions

In [2]:
```
## Aux function

def fshape(dataframe1):
    print('Shape : ', dataframe1.count(), len(dataframe1.columns))

def fhead(dataframe1, num_records=3):
        ## Show all columns - pandas dataframe
        # import pandas as pd
        # pd.options.display.max_columns = None

    return dataframe1.limit(num_records).toPandas()

def fsummary(dataframe1):
    return dataframe1.summary().toPandas()
```

```
Calculation started (calculation_id=1cc38d12-5649-59c6-b5b8-5e556f4a7102) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
```

Quick info related to the dataset

Original dataset - converted to Parquet (typical file format stored in S3)

- https://archive.ics.uci.edu/ml/datasets/online+retail

In [3]:
```
## read local file
input_location = 's3://...S3...BUCKET...NAME/s3_data/input/OnlineRetail__AWS.parquet'
sdf = spark.read.parquet(input_location)
# sdf.printSchema()

fshape(sdf)
fhead(sdf)
```

```
Calculation started (calculation_id=52c38d12-5766-68cb-30c2-6b84c752dd39) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
```

```
Calculation completed.
Shape :  541909 8
  InvoiceNo StockCode  ... CustomerID         Country
0    536365    85123A  ...    17850.0  United Kingdom
1    536365     71053  ...    17850.0  United Kingdom
2    536365    84406B  ...    17850.0  United Kingdom

[3 rows x 8 columns]
```

Create dataset with customer purchase history and apply CLV formula

- customer_id
- invoice_date
- revenue : monetary value

In [4]:
```
sdf.createOrReplaceTempView('TB_SALES_SDF')
spark.sql('select max(TO_DATE(InvoiceDate)) as current_date_for_FRMV_CLV, current_date as not_today from TB_SALES_SDF').show()
```

```
Calculation started (calculation_id=eec38d12-70c3-7d8d-b49a-baa746f0547f) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
+-------------------------+----------+
|current_date_for_FRMV_CLV| not_today|
+-------------------------+----------+
|               2011-12-09|2023-03-25|
+-------------------------+----------+
```

Information to understand the formula

The formula to calculates: **Customer Lifetime Value (CLV) using the FRM (Frequency, Recency, Monetary Value) approach with a discount rate of 10%** .

- monetary_value: the total monetary value spent by the customer.
- frequency: the frequency of customer purchases, i.e., how many times they made a purchase.
- recency_dt: the recency of the customer's purchases, i.e., how many days ago they made their last purchase.
- 365: the number of days in a year.
- 0.1: the discount rate used to calculate the present value of future cash flows.

The formula itself consists of three parts:

- (monetary_value / frequency): this part calculates the average value of each purchase made by the customer.
- (1 - ((recency + 1) / 365)): this part calculates the probability of the customer returning to make a purchase based on the time since their last purchase. The longer the time since the last purchase, the lower the probability of the customer returning to make a purchase.
- / (1 + discount): this part applies the discount rate to calculate the present value of future cash flows.

In [5]:
```
## formula to calculate CLV
def fnc_customer_clv_udf(monetary_value_f, frequency_f, recency_f, discount_f=0.1):
    return round ( ( (monetary_value_f / frequency_f) * (1 - ((recency_f + 1) / 365)) / (1 + discount_f) ) , 2)

## Register the formula to be used by Spark-SQL
from pyspark.sql.types import FloatType

spark.udf.register('fnc_customer_clv_udf', fnc_customer_clv_udf, FloatType())

print("Catalog Entry:")
[print(r) for r in spark.catalog.listFunctions() if "fnc_customer_clv_udf" in r.name]
```

```
Calculation started (calculation_id=1ec38d12-75f6-a153-eaf1-4f901cbb4630) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
Catalog Entry:
Function(name='fnc_customer_clv_udf', description=None, className='org.apache.spark.sql.UDFRegistration$$Lambda$4913/1436708388', isTemporary=True)
[None]
```

In [6]:

```
## Apply some filters and create the main customer purchase history as an example
sql_query_clv = """
WITH TB_SALES_V AS
(
    SELECT CustomerID as customer_id
        , COUNT(DISTINCT (InvoiceDate))  as frequency
        , DATEDIFF( current_date , MAX (InvoiceDate) )  as recency_now
        , ROUND(SUM(Quantity * UnitPrice), 2) as monetary_value
        , ROUND(avg(Quantity * UnitPrice), 2) as avg_revenue
        , MIN(InvoiceDate) as dt_first_Invoice
        , MAX(InvoiceDate) as dt_last_Invoice
        -- , ROUND(AVG(Quantity), 2) as avg_items
        -- , ROUND(SUM(Quantity), 2) as total_items
    FROM TB_SALES_SDF
    WHERE 1 = 1
        AND InvoiceDate IS NOT NULL
        AND Quantity > 0
        AND UnitPrice > 0
    GROUP BY customer_id
)
SELECT tb3.*
  , ROUND ( ( (monetary_value / frequency) * (1 - ((recency_dt + 1) / 365)) / (1 + 0.1) ) , 2) AS CLV_SQL -- discount of 0.1
  , fnc_customer_clv_udf(monetary_value,frequency,recency_dt) AS CLV_UDF
FROM (
    SELECT tb1.*
        , CAST( DATEDIFF(tb2.dt_current_date , tb1.dt_last_Invoice ) as float) as recency_dt
    FROM TB_SALES_V as tb1
    CROSS JOIN (SELECT MAX(dt_last_Invoice) AS dt_current_date FROM TB_SALES_V) tb2
    ) tb3
WHERE 1 = 1
  AND monetary_value > 0
  AND frequency > 0
  AND customer_id IS NOT NULL
ORDER BY monetary_value DESC
"""

sdf_clv = spark.sql(sql_query_clv)
sdf_clv.printSchema()
```

```
Calculation started (calculation_id=88c38d12-8f47-121b-3b46-2dcda79b09d0) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
root
 |-- customer_id: double (nullable = true)
 |-- frequency: long (nullable = false)
 |-- recency_now: integer (nullable = true)
 |-- monetary_value: double (nullable = true)
 |-- avg_revenue: double (nullable = true)
 |-- dt_first_Invoice: timestamp (nullable = true)
 |-- dt_last_Invoice: timestamp (nullable = true)
 |-- recency_dt: float (nullable = true)
 |-- CLV_SQL: double (nullable = true)
 |-- CLV_UDF: float (nullable = true)
```

In [7]:
```
print('clv_SQL and clv_udf provide the same information - just show how to implement it using 2 solutions... SQL and UDF')
fhead(sdf_clv)
```

```
Calculation started (calculation_id=b2c38d12-9493-1b32-5a86-66d8b9f4037e) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
clv_SQL and clv_udf provide the same information - just show how to implement it using 2 solutions... SQL and UDF
   customer_id  frequency  recency_now  ...  recency_dt   CLV_SQL       CLV_UDF
0      14646.0         51         4125  ...         1.0   3555.12   3555.120117
1      16446.0          2         4124  ...         0.0  76368.60  76368.601562
2      17450.0         27         4134  ...        10.0   3961.80   3961.800049

[3 rows x 10 columns]
```

Machine Learning - Customer segmentation and plot

- Predictive Power (KI) = 0.741 and Prediction Confidence (KR) = 0.917

In [8]:
```
sdf_clv.createOrReplaceTempView('TB_CLV_SDF')
```

```
Calculation started (calculation_id=8cc38d12-adfe-99c7-e21f-51546f1d63f8) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
```

In [ ]:
```

```

In [9]:
```
def ml_sql_prediction(filename1='./CLV_AWS__Spark_Execution_v1.sql'):
#       text_rdd = sc.textFile(filename1)
        # concatenate all lines into a single STRING    ** obs. incluir um tab em todas as linhas via Notepad
#       text_sql_ml = text_rdd.reduce(lambda x, y: x + y)

        text_sql_ml2 = f"""

                 SELECT TB_CLV_SDF.*,
                ( CASE
                WHEN ( ( ( `frequency`  > 1.0e1 AND `frequency`  <= 1.14e2 ) ) ) THEN 9
                WHEN ( ((abs(year(`dt_first_Invoice`) - 2.01e3) <= 10e-9) OR ( (`dt_first_Invoice` IS NULL ) ) ) AND ((abs(`frequency` - 1.0e0) <= 10e-9) OR (abs(`frequency` - 2.0e0) <= 10e-9) C
                WHEN ( ((abs(`frequency` - 7.0e0) <= 10e-9) OR  ( `frequency`  >= 8.0e0 AND `frequency`  <= 1.3e1 ) ) ) THEN 3
                WHEN ( ( ( `recency_dt`  >= 0.0e0 AND `recency_dt`  <= 4.0e0 ) ) ) THEN 10
                WHEN ( ( ( `CLV_SQL`  > 9.0245000000000005e2 AND `CLV_SQL`  <= 7.49729e3 ) ) ) THEN 6
                WHEN ( ( ( `CLV_SQL`  > 3.8501999999999998e2 AND `CLV_SQL`  <= 9.0245000000000005e2 ) ) AND ((abs(`frequency` - 1.0e0) <= 10e-9) OR (abs(`frequency` - 2.0e0) <= 10e-9) OR (abs(`f
                WHEN ( ( ( (datediff(concat(year(`dt_first_Invoice`),'-',month(`dt_first_Invoice`),'-',day(`dt_first_Invoice`)),concat(year(`dt_first_Invoice`),'-01-01')) + 1)  > 1.3e1 AND (date
                WHEN ( ((abs(month(`dt_last_Invoice`) - 3.0e0) <= 10e-9) OR (abs(month(`dt_last_Invoice`) - 4.0e0) <= 10e-9) OR (abs(month(`dt_last_Invoice`) - 5.0e0) <= 10e-9) OR (abs(month(`dt
                WHEN ( ( ( `recency_dt`  >= 3.0e0 AND `recency_dt`  <= 2.5e1 ) OR  ( `recency_dt`  > 3.1e1 AND `recency_dt`  <= 3.6e1 ) OR  ( `recency_dt`  > 3.25e2 AND `recency_dt`  <= 3.74e2 )
                WHEN ( ( ( (datediff(concat(year(`dt_last_Invoice`),'-',month(`dt_last_Invoice`),'-',day(`dt_last_Invoice`)),concat(year(`dt_last_Invoice`),'-01-01')) + 1)  >= 4.0e0 AND (datedif
                ELSE 11
                END ) AS kc_monetary_value
                FROM TB_CLV_SDF

                """

        return text_sql_ml2
```

```
Calculation started (calculation_id=38c38d12-b350-e485-b152-0c1959a654fb) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
```

In [2]:
```
# ml_sql_prediction()
```

In [10]:
```
ml_spark = ml_sql_prediction()

sdf_ml = spark.sql(ml_spark)

sdf_ml.printSchema()
# fhead(sdf_ml)
sdf_ml.show(3, vertical=True)
```

```
Calculation started (calculation_id=c2c38d12-b88d-4aa6-7031-af3533731feb) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
root
 |-- customer_id: double (nullable = true)
 |-- frequency: long (nullable = false)
 |-- recency_now: integer (nullable = true)
 |-- monetary_value: double (nullable = true)
 |-- avg_revenue: double (nullable = true)
 |-- dt_first_Invoice: timestamp (nullable = true)
 |-- dt_last_Invoice: timestamp (nullable = true)
 |-- recency_dt: float (nullable = true)
 |-- CLV_SQL: double (nullable = true)
 |-- CLV_UDF: float (nullable = true)
 |-- kc_monetary_value: integer (nullable = false)

-RECORD 0--------------------------------
 customer_id       | 14646.0
 frequency         | 51
 recency_now       | 4125
 monetary_value    | 200541.0
 avg_revenue       | 137.36
 dt_first_Invoice  | 2010-12-20 10:09:00
 dt_last_Invoice   | 2011-12-08 00:12:00
 recency_dt        | 1.0
 CLV_SQL           | 3555.12
 CLV_UDF           | 3555.12
 kc_monetary_value | 9
-RECORD 1--------------------------------
 customer_id       | 16446.0
 frequency         | 2
 recency_now       | 4124
 monetary_value    | 168472.49
 avg_revenue       | 56157.5
 dt_first_Invoice  | 2011-05-18 09:52:00
 dt_last_Invoice   | 2011-12-09 09:15:00
 recency_dt        | 0.0
 CLV_SQL           | 76368.6
 CLV_UDF           | 76368.6
 kc_monetary_value | 10
-RECORD 2--------------------------------
 customer_id       | 17450.0
 frequency         | 27
 recency_now       | 4134
 monetary_value    | 121321.71
 avg_revenue       | 588.94
 dt_first_Invoice  | 2010-12-07 09:23:00
 dt_last_Invoice   | 2011-11-29 09:56:00
 recency_dt        | 10.0
 CLV_SQL           | 3961.8
 CLV_UDF           | 3961.8
 kc_monetary_value | 9
only showing top 3 rows
```

```
In [11]: fhead(sdf_ml,num_records=4)
```

```
Calculation started (calculation_id=fec38d12-d1ad-f99f-cb56-b686a9f1d347) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
Calculation completed.
   customer_id  frequency  ...       CLV_UDF  kc_monetary_value
0      14646.0         51  ...   3555.120117                  9
1      16446.0          2  ...  76368.601562                 10
2      17450.0         27  ...   3961.800049                  9
3      18102.0         30  ...   3362.270020                  9

[4 rows x 11 columns]
```

```
In [3]: ## Export as parquet file
        # sdf_ml.write.mode('overwrite').parquet(output_file)
```

Plot and Report sample

```
In [12]: sdf_ml.createOrReplaceTempView('TB_CLV_SDF_ML')

ml_rpt_sql = """
WITH TB_CLUSTER AS
(
    select kc_monetary_value as cluster_number
    , count(distinct customer_id) as customer_count
    , avg(clv_sql) avg_clv
    , avg(monetary_value) avg_monetary_value
    -- , count(*) as qty_records
    FROM TB_CLV_SDF_ML
    group by kc_monetary_value
)
SELECT cluster_number
--     , customer_count
    , ROUND( customer_count / (select sum(customer_count) from TB_CLUSTER ) * 100, 2) as percent_of_customers
    , ROUND( avg_clv, 2) as avg_clv
    , ROUND( avg_monetary_value, 2) as avg_monetary_value
FROM TB_CLUSTER tb1
order by avg_clv desc
"""

sdf_ml_rpt = spark.sql(ml_rpt_sql)
# sdf_ml_rpt.printSchema()
sdf_ml_rpt.show()
```

```
Calculation started (calculation_id=78c38d12-eac9-f874-2363-3f42099781af) in (session=54c38d11-bc47-e5c7-19b9-cd14cdf24b9b). Checking calculation status...
Progress:   0%|          |elapsed time = 00:00s
```

```
Calculation completed.
+--------------+--------------------+-------+------------------+
|cluster_number|percent_of_customers|avg_clv|avg_monetary_value|
+--------------+--------------------+-------+------------------+
|             6|                1.41|1627.96|           6390.09|
|            10|                4.78| 859.47|           2221.99|
|             9|                3.88| 607.03|          15079.84|
|             2|               10.07| 543.86|           1592.38|
|             3|                5.45| 359.51|           3559.05|
|             1|                8.47|  201.7|           1180.36|
|             5|               14.02| 181.48|             579.6|
|             4|               33.19| 156.01|            519.58|
|             7|               10.07| 145.49|            342.34|
|             8|                8.66|  126.8|            529.09|
+--------------+--------------------+-------+------------------+
```

Plot

In [4]:
```
# sdf_ml_rpt.pandas_api().plot.scatter(x='avg_monetary_value', y='avg_clv', size='percent_of_customers',
#                                       title='Customer value_clv vs Monetary value (FRM -frequency, recency, monetary value)',
#                                       color='cluster_number')
```

Optimization in Spark - considerations

Spark 1.x : Catalyst Optimizer and Tungsten Project (CPU, cache and memoery efficiency, eliminating the overhead of JVM objects and garbage collection)

Spark 2.x : Cost-Based Optimizer (CBO) to improve queries with multiple joins, using table statistics to determine the most efficient query execution plan

Spark 3.x : Adaptive Query Execution (AQE) is an optimization technique in Spark SQL that use runtime statistics to choose the most eficient query execution plan, which is enabled by default since Apache Spark 3.2.0

- https://spark.apache.org/docs/latest/sql-performance-tuning.html
- three major features in AQE: including coalescing post-shuffle partitions, converting sort-merge join to broadcast join, and skew join optimization

This notebook use Spark 3.x and Adaptive Query Execution (AQE)

In [2]:
```
!jupyter nbconvert --to html Customer_Value_demo__using_Athena-Spark-Job_v1_20230325.ipynb
```

In [ ]:
```
```