UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Mohga Soliman Emam Hewashy

Financial Fraud Detection: A Declarative Approach

Master's Thesis (30 ECTS)

Supervisor(s): Ahmed Awad, PhD

Tartu 2023

Financial Fraud Detection: A Declarative Approach

Abstract:
The aim of the thesis is to introduce a declarative approach for the statically specified financial fraud detection use cases and scenarios defined by the financial regulatory entities to capture money laundering and terrorist financing activities ML/TF. The thesis introduces the Match_Recognize Python library that replaces the static rules ingested into the financial institutions and organizations transaction monitoring system to detect financial fraudulence and suspicious activities. The introduced Match_Recognize Python library mimics the functionality of the SQL Match_Recognize clause which performs pattern recognition using regular expressions which can be used to detect financial fraud patterns and therefore eliminate the need to design and develop dedicated static use case scenarios. Using the Match_Recognize library, financial institutions and organizations can produce the financial fraud detection use case scenarios required by the financial regulatory entities using simple regular expressions that are passed to the library alongside the dataset. Additionally, the Match_Recognize Python library contains a Match_Recognize Automaton function that validates new, proposed patterns in the form of regular expressions within the Match_Recognize clause pattern regular expression by using the non-deterministic Automaton created dynamically from the Match_Recognize pattern regular expression. The thesis also introduces versatile, pliant financial fraud detection scenarios inspired by the Financial Action Task Force Recommendations. Evaluation of the Match_Recognize Python library is conducted by running the financial fraud detection scenarios on both the Match_Recognize clause in Oracle database and Match_Recognize Python Library then comparing the results. A dedicated time log has been created in order to compare the averaged time taken to simulate each financial fraud detection scenario statically and to simulate it using the Match_Recognize Python library. The results show that indeed the Match_Recognize library reduces the financial fraud scenario simulations time by 96.3%.
Keywords:
Pattern Matching, Match_Recognize, Python, Financial Fraud Detection, Anti-Money Laundry, Counter Terrorism Financing, Nondeterministic Automaton, Suspicious Activity Monitoring.

CERCS: P170 Computer science, numerical analysis, systems, control. S181 Financial science.

Finantspettuste avastamine: deklaratiivne lähenemine

Lühikokkuvõte:

Lõputöö eesmärk on tutvustada deklaratiivset lähenemist finantsjärelevalve üksuste poolt määratletud staatiliselt määratletud finantspettuste avastamise kasutusjuhtudele ja stsenaariumidele, et tabada rahapesu ja terrorismi rahastamisega seotud tegevusi ML/TF. Lõputöö tutvustab Match_Recognize Pythoni teeki, mis asendab pankade ja finantsasutuste tehingute jälgimise süsteemi sisestatud staatilisi reegleid, et tuvastada finantspettusi ja kahtlasi tegevusi. Kasutusele võetud Match_Recognize Pythoni teek jäljendab SQL-i klausli Match_Recognize funktsionaalsust, mis teostab mustrituvastuse regulaaravaldiste abil, mida saab kasutada finantspettuste mustrite tuvastamiseks ja seega välistab vajaduse kavandada ja arendada spetsiaalseid staatiliste kasutusjuhtumite stsenaariume. Teeki Match_Recognize kasutades saavad finantsasutused ja organisatsioonid koostada finantspettuste tuvastamise kasutusjuhtumi stsenaariume, mida nõuavad finantsregulatsiooniüksused, kasutades lihtsaid regulaaravaldisi, mis edastatakse koos andmestikuga teeki.

Lisaks sisaldab Match_Recognize Pythoni teek funktsiooni Match_Recognize Automaton, mis kinnitab uued pakutud mustrid regulaaravaldiste kujul Match_Recognize klausli mustri regulaaravaldises, kasutades mittedeterministlikku automaati, mis on loodud dünaamiliselt Match_Recognize mustri regulaaravaldisest. Lõputöö tutvustab ka mitmeid mitmekülgseid ja paindlikke finantspettuste avastamise stsenaariume, mis on inspireeritud Financial Action Task Force'i soovitustest. Match_Recognize Pythoni teegi hindamine viiakse läbi, käivitades finantspettuste tuvastamise stsenaariumid nii Oracle'i andmebaasi Match_Recognize klausli kui ka Match_Recognize Pythoni raamatukogu ja seejärel tulemusi võrreldes. Spetsiaalne ajalogi on loodud selleks, et võrrelda iga finantspettuste tuvastamise stsenaariumi staatiliseks simuleerimiseks kuluvat keskmist aega ja seda Match_Recognize Pythoni teegi abil. Tulemused näitavad, et Match_Recognize teek vähendab tõepoolest finantspettuste stsenaariumi simulatsiooni aega 96.3%.

Võtmesõnad:

Mustri sobitamine, Match_Recognize, Python, finantspettuste tuvastamine, rahapesuvastane võitlus, terrorismivastane rahastamine, mittedeterministlik automaatika, kahtlase tegevuse jälgimine.

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria). S181 Rahandus.

Contents

1 Introduction

The thesis content inspired a dedicated book chapter [FRDA] built entirely on the work presented in the thesis in one of the most established financial fraud and corruption textbook series. The chapter [FRDA] is under review at the time of authorizing the thesis and expected to be published later this year.

Financial Fraud [DFKRJ12], Money Laundering[mon], and Terrorist Financing [fin] are forms of financial crime that affect financial institutions and organizations all over the world.

It is estimated that every bank in the UK has been fined at least once due to weak controls in Anti-Fraud, and Anti Money Laundering/Counter Terrorist Financing (AML/CTF) in the past decade [CSD]. Meaning that these financial institutions and organizations were found guilty due to incompetence in detecting suspicious activity and prevention of money laundering and terrorist financing detection.

Financial institutions and organizations are able detect fraudulent activities using predefined pattern recognition [Rep18]

Financial regulatory entities [reg22] such as the Financial Intelligence Unit (FIU) and Financial Action Task Force (FATF) specify several suspicious activity patterns to provide the minimum requirement for financial institutions and organizations to detect Fraudulence, Money Laundering and Terrorist Financing activities and based on those patterns the financial regulatory entities perform a regular yearly inspection on the financial institutions and organizations. The financial institutions and organizations are required to design and develop their own interpretations of the suspicious activity patterns specified by the financial regulatory entities as well as incorporating them into their monitoring processes to detect any financial crime activities. However, when financial institutions and organizations get fined it means they have failed at least once in the implementation of these Anti-Fraud, and AML/CTF detection patterns.

Getting fined is catastrophic for financial institutions and organizations because it ruins their reputation and potentially loses the customers' trust.

The only two options financial institutions and organizations must ensure compliance and avoid getting fined is to inject the rules specified by the financial regulatory entities into the dedicated monitoring system to successfully detect any fraudulent activities. The two options are:

1. Design and develop dedicated use case scenarios that mimic the statically specified financial fraud patterns to ingest the use case scenarios into the financial institutions and organizations' Transaction Monitoring system [RMP17].

2. Translate the specified financial fraud patterns into complex regular expressions that run directly on the financial institutions and organizations' database to detect suspicious activities.

The first option is not considered to be an optimal choice since it assumes that the financial institutions and organizations have already implemented all of the Anti-Fraud, and AML/CTF detection and prevention controls mentioned by the financial regulatory entities which isn't enough to stop criminal activities in real life due to the fact that as time progresses, criminals get more and more intelligent and learn how to abuse the financial institutions and organizations controls to the extent that the criminals come up with new patterns and typologies to defeat the monitoring system and overcome the financial institutions and organizations Anti-Fraud, and AML/CTF detection and prevention controls.
Due to the above reasons, financial institutions and organizations must update their Anti-Fraud and AML/CTF detection controls to prevent criminal activity regularly. However, financial institutions and organizations use static detection controls to monitor criminals' activities, meaning that when criminals invent new ways to commit financial crime, financial institutions and organizations have no defense against such activities until the possible threat is identified and the process of creating a new dedicated control begins to take place.
Therefore, there is a risk of having a continuous breach in the Anti-Fraud, and AML/CTF detection and prevention controls and the financial institutions and organizations not able to detect it nor prevent it in time and then the breach indeed will be overlooked by the monitoring system and hence there will always be a degree of incompetence in the financial institutions and organizations Anti-Fraud, and AML/CTF detection and prevention controls. Hence using static rules is proven useless in detecting Fraudulence, Money Laundering/Terrorist Financing (ML/TF) schemes overtime [ZSF17].

Another caveat for the first option is that when the financial regulatory entities specify suspicious and criminal activities, it is shared it in the form of static patterns[sus]:

1. Customers deposit cash by means of numerous credit slips so that the total of each deposit is unremarkable, but the total of all the credits is significant.

2. Clients with no discernible reason for using the firm's service, e.g., clients with distant addresses who could find the same services nearer their home base; clients whose requirements are not in the normal pattern of the firm's business which could be more easily serviced elsewhere.

3. Number of transactions by the same counterparty in insignificant amounts of the same security, each purchased for cash and then sold in one transaction, the proceeds being credited to an account different from the original account.

Accordingly, financial institutions and organizations tend to create the static pattern recognition rules and use case scenarios to detect each Fraud, and ML/TF activity pattern specified by the financial regulatory entities based on their own interpretation and alignment with the existent Anti-Fraud controls. However, criminals are getting more intelligent by the second and static scenario detection will not hold against the test of time and the ever-changing criminal schemes.

That leaves the financial institutions and organizations with the second option which is translating the specified rules and use case scenarios [Nah19] into complex regular expressions and simply use those on the database. This is an effective and optimal option since the regular expressions that represent the rules and use case scenarios will run directly on the database which will allow the financial institutions and organizations to avoid the problem of having a fatal failure point which is the layer containing the rules and use case scenarios in option one.

However, to understand where to ingest the second option into the monitoring system, a brief description of the Anti-Fraud, and AML/CTF detection control pipeline is introduced in the following phases:

1. The customer data loads, during this phase the financial institutions and organizations do ETL.

2. During the customer on-boarding phase, sanction screening and risk assessment are implemented on each customer to ensure that no customers are on any sanction lists.

3. After that the customer data is processed into the system.

4. During the transaction monitoring phase all the customers data is run across the monitoring engine where this engine is fed with static pattern recognition scenarios and rules to identify suspicious activities.

5. The AML Operation phase starts, where professionals study the cases and get the final word whether the identified activities are indeed fraudulent or not.

6. Finally, the process of generating reports with the customers' activities that the professionals deemed to be suspicious.

Clauses such as Match_Recognize can identify and recognize patterns swiftly using simple regular expressions [BTC06].

Match_Recognize [ISO] uses a *PATTERN* sub-clause that specifies the pattern to be matched as a regular expression over one or more correlation variables.
This gives the financial institutions and organizations the ability to replace the static rules and use case scenarios with declarative, simple Match_Recognize queries.
The main reason financial institutions and organizations are unable to implement Match_Recognize directly on the datasets, is that Match_Recognize is not supported in many database management systems [Pet22a]. Even the Oracle database has numerous problems supporting Match_Recognize clause [JDB]. Therefore, the thesis introduces the Match_Recognize Python library to support pattern recognition by emulating the Match_Recognize functionality and will be used to replace the static use case scenarios by simple regular expressions.
A famous Anti-Fraud, and AML/CTF detection pattern which is inspired from the FATF 40 Recommendations [FAT] is the U-shaped transactions which is one of the most well-known suspicious activity patterns. The pattern starts with identifying high transaction amount volume, followed by a rapid decrease in the transaction amount volume then by a quick increase. Each financial institution and organization specify whether the pattern is detecting a dedicated, and specific type of transactions such as overseas deposits, incoming transfers, etc.
The Match_Recognize Python library supports the usage of the SQL Match_Recognize clause, allowing users to simply install and use the library with any pattern, definition, dataset required.
A simulation of an Anti-Fraud, and AML/CTF use case scenario is presented in the below example identifying customers with transaction amount volumes with the "U" shape implemented by the SQL Match_Recognize clause.

```
SELECT
        *
    FROM
        Transactions
    MATCH_RECOGNIZE    (
```

```
        MEASURES
                customer_id,
                f i r s t ( timestamp )         as
                begin_time , last ( timestamp )       as
                end_time
          PATTERN(UP+ DOWN+      UP+)
        DEFINE
                UP as transaction_amount <=prev ( transaction_amount) ,
               DOWN transaction_amount >=prev ( transaction_amount )
    )   as   T
```

1.1 Problem Statement

Pattern recognition using SQL Match_Recognize has limited to no support for most of the database management systems used by the financial institutions and organizations preventing the financial institutions and organizations from using the SQL Match_Recognize clause, as well as no support for the Python programming language which is the most famous, preferred, supported language for data analysis and science within the financial institutions and organizations.

Python is still considered to have an incredibly limited support for powerful pattern recognition clauses such as Match_Recognize even with the increasing support for Match_Recognize such as the recent Match_Recognize clause support in RSSQL [Ful]. Even though, Python has been the most used programming language in the data science community for at least a decade now, and was created 31 years ago in 1991, It has not matured enough its pattern recognition functionalities to reach its maximum capabilities like other programming languages. Instead of using pattern recognition clauses such as Match_Recognize, financial institutions and organizations need to design, develop, and implement static use case scenarios to detect ML/TF behaviors which is very inefficient to the financial institutions and organizations as discussed in the introduction section. The alternative to using both the static use case scenarios in recognizing the ML/TF behaviors is to use simple regular expressions that can recognize the specified patterns and extracting the data stream that matches with the ML/TF behaviors.

Regular expressions are protected against becoming obsolete against the ever-changing criminal schemes because regular expressions are easily customizable, flexible in pattern recognition, resource efficient and decrease the time spent on design, develop, and implement new use case scenarios drastically. Regular expressions can be easily manipulated to create powerful use case scenarios that recognize ML/TF behaviors and are flexible enough that it can be used individually as a monitoring engine or integrated into an existing one. Regular expressions can be used to develop applications that forecast criminal behaviors by combining it with Machine Learning or Neural Networks. However, we do not explore the forecasting capabilities of integrating regular expressions with such techniques.

2 Background

The background section discusses the structure of Match_Recognize clause and the Match_Recognize building blocks as well as the design and development of the Match_Recognize Python library Automaton in more detail.

2.1 Match_Recognize

The Match_Recognize clause performs pattern recognition with the assistance of regular expressions. It takes the regular expression alongside several sub-clauses and outputs a stream [Kör21] of data matching with the corresponding defined pattern.
The Match_Recognize [Rep18] clause has a unique structure that is different from the normal queries. It consists of the following sub-clauses:

1. MEASURES: Returns the dataset attribute of the successfully matched with the pattern events. The MEASURES clause also specifies how the output is returned. The screenshot below Fig. 1 presents a brief example of the MEASURES subclause. The MEASURES sub-clause returns the first occurrence of the TICKER dataset timestamp and its corresponding price [Lak17].

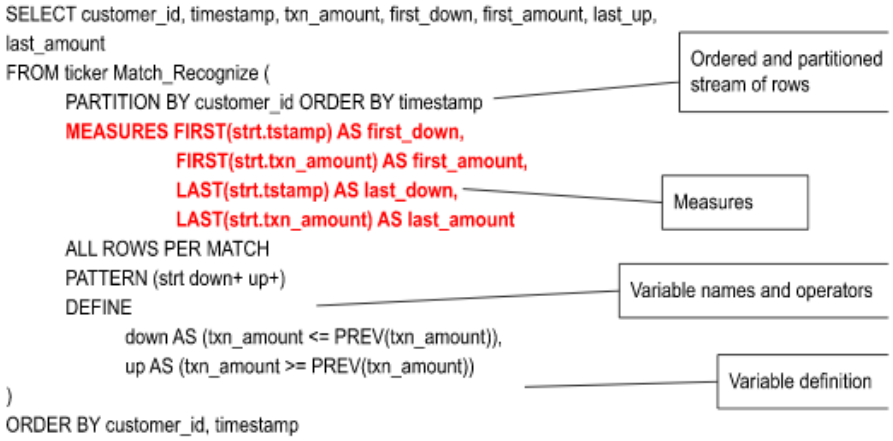

Figure 1. measures sub-clause example.

2. PATTERN: Specifies the pattern to be matched using regular expressions inside the parentheses. The example below in Fig. 2 presents the PATTERN sub-clause stating the recognizable pattern in the form of regular expressions.
   The following regular expression *(strt down+ up+)* specifies a pattern from the start point followed by a decrease, then followed with an increase [Lak17].

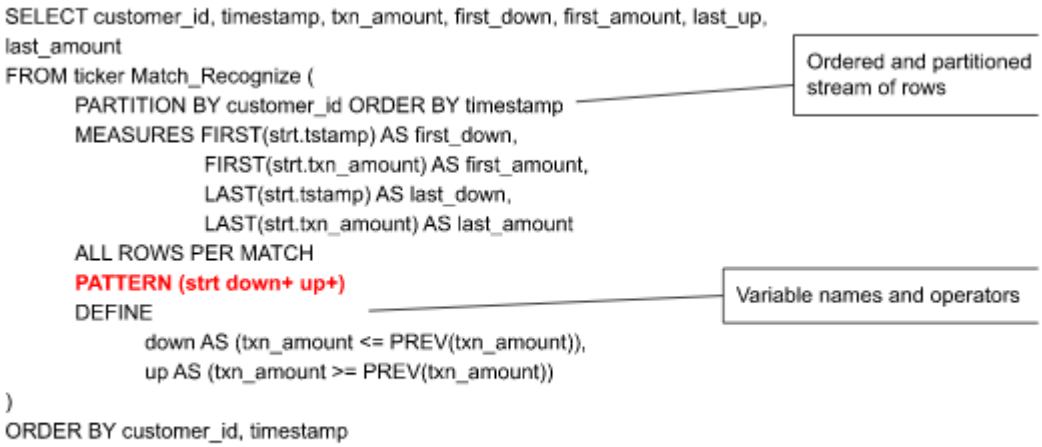

Figure 2. pattern sub-clause example.

3. DEFINE: States the expressions corresponding to regular expression variables mentioned in the PATTERN clause. The example presented in Fig. 3 presents the DEFINE sub-clause which contains the expressions corresponding to each regular expression symbol. The down symbol corresponds to the pattern of when the current price is lower than or equal to the previous price. Meanwhile, the up symbol corresponds to the pattern of when the current price is bigger than or equal to the previous price. up and down patterns are mentioned in [BH20] chapter with more detail.
4. PARTITION BY: Specifies the stream attribute which divides the dataset input streams. The example shown in Fig. 4 specifies that the input stream will be divided by the symbol data attribute. Meaning that pattern recognition will be implemented for each pattern at a time.

5. ORDER BY: States the specific order of the partitioned stream. The example below indicates that the output stream will be ordered based on the tstamp data attribute.

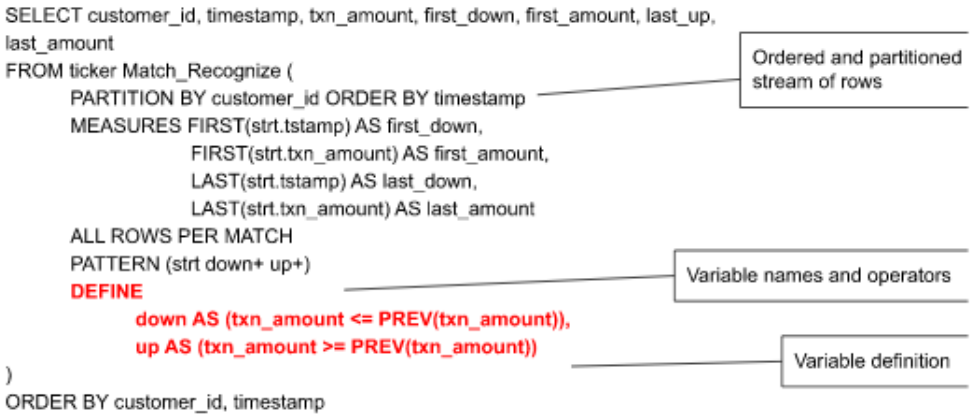

Figure 3. define sub-clause example [Lak17].

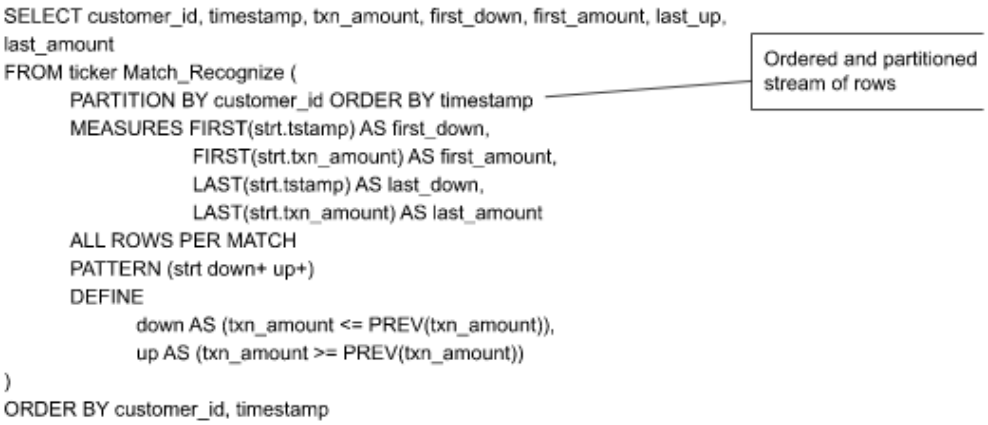

Figure 4. partition by and order by sub-clauses example [Lak17].

2.2 Match Automaton

The Match_Recognize Python library introduces a Match_Recognize Automaton function called the Match_Automaton. The Match_Automaton function accepts two inputs; the first input is the Match_Recognize clause and the second input is the new pattern in the form of regular expression. The Match_Automaton function is dedicated for creating a dynamic automaton that takes the Match_Recognize clause specified pattern then dynamically generates a non-deterministic automaton [Bro89] with dynamic input symbols, input streams, transition tables to validate the new proposed pattern against it.

It is important to note that the automaton developed in the library is a Nondeterministic Finite Automaton NFA not a Deterministic Finite Automaton DFA [Bro89].

To understand the reason behind designing and developing the library function using Non-Deterministic Finite Automaton rather than using Deterministic Finite Automaton, a brief comparison between the Deterministic and Non-Deterministic Finite Automaton is shown within the table 1:

Deterministic Finite Automaton	Non-Deterministic Finite Automaton
More RAM allocation is required.	Less RAM allocation is required.
On an input, we reach a deterministic and unique state for a particular state.	On an input, we can reach more than one state for a given state.
For each transition in the transition able one state is called deterministic.	For each transition in the transition table a subset of states is called non-deterministic.
In DFA, backtracking is permitted.	In NFA, it is not always possible to backtrack.
The Empty String transition is not available in DFA.	The Empty String transition is indeed available in NFA.

Table 1. Difference between DFA and NFA Automaton.

Unlike pattern matching, pattern recognition compares two or more patterns to get the most likely match, not the exact match. Since the Match_Recognize clause is a pattern recognition clause not a pattern matching clause then the automaton needs to be able to find the match for the not the exact match.

Pattern Matching [mat22] is identified as checking a given pattern for the presence of the constituents of some pattern to find the exact match [HP03]. Meanwhile, Pattern Recognition [rec22] is identified similarly to pattern matching, However, the output will find the matches for the match and not necessarily an exact match [CC13].
Using the NFA automaton users can get more than one state from using each input with the given states, achieving the and not exact match concept of the Match_Recognize clause. Also, the NFA automaton is design to support regular expressions used in the Match_Recognize PATTERN sub-clause, such as:

1. (DOWN+ FLAT* DOWN+)
2. (NORMAL+ UNUSUAL* NORMAL+)

Additionally, the (*) quantifier matches zero occurrences or more. Meaning that the automaton needs to match the FLAT* symbol to both zero and more occurrence and the NFA automaton is the automaton which accepts empty string transitions not the DFA automaton.

2.3 Goal

The goal of the thesis is to design and develop an open-source Python library to support pattern recognition using Match_Recognize. Static pattern recognition scenarios are not reliable nor functional enough against the ever-changing financial fraud schemes. That is why supporting pattern recognition using Match_Recognize is the optimal solution because it enables the users to build powerful, declarative pattern recognition rules and use case scenarios that are reliable against the Fraud, and ML/TF ever changing patterns and typologies. The latest version of Python at the time of drafting this thesis, is Python 3.9, yet it has an extremely naive approach of handling pattern recognition.
The Match_Recognize Python library will replace the static patterns inside the transactions monitoring engine in Fig. 5, allowing the user to specify Anti-Fraud, and AML/CTF pattern recognition controls.
The idea of using Match_Recognize as a row processing engine was first introduced in [Han] as a state-of-the-art usage of the powerful pattern recognition clause and it is also used for pattern recognition within the CEP engine in [AS17].
Based on the pipeline mentioned in the introduction section, it can be concluded that the Match_Recognize Python library will be able to replace the transaction monitoring engine [AGAG21] in Fig. 5 as it uses declarative pattern recognition rather than the static

rules and scenarios transaction monitoring engine. The financial institution and organization data will go through the Match_Recognize Python library to allow the user to generate declarative pattern recognition scenarios without consuming time and resources in designing and developing static, individual use scenarios.

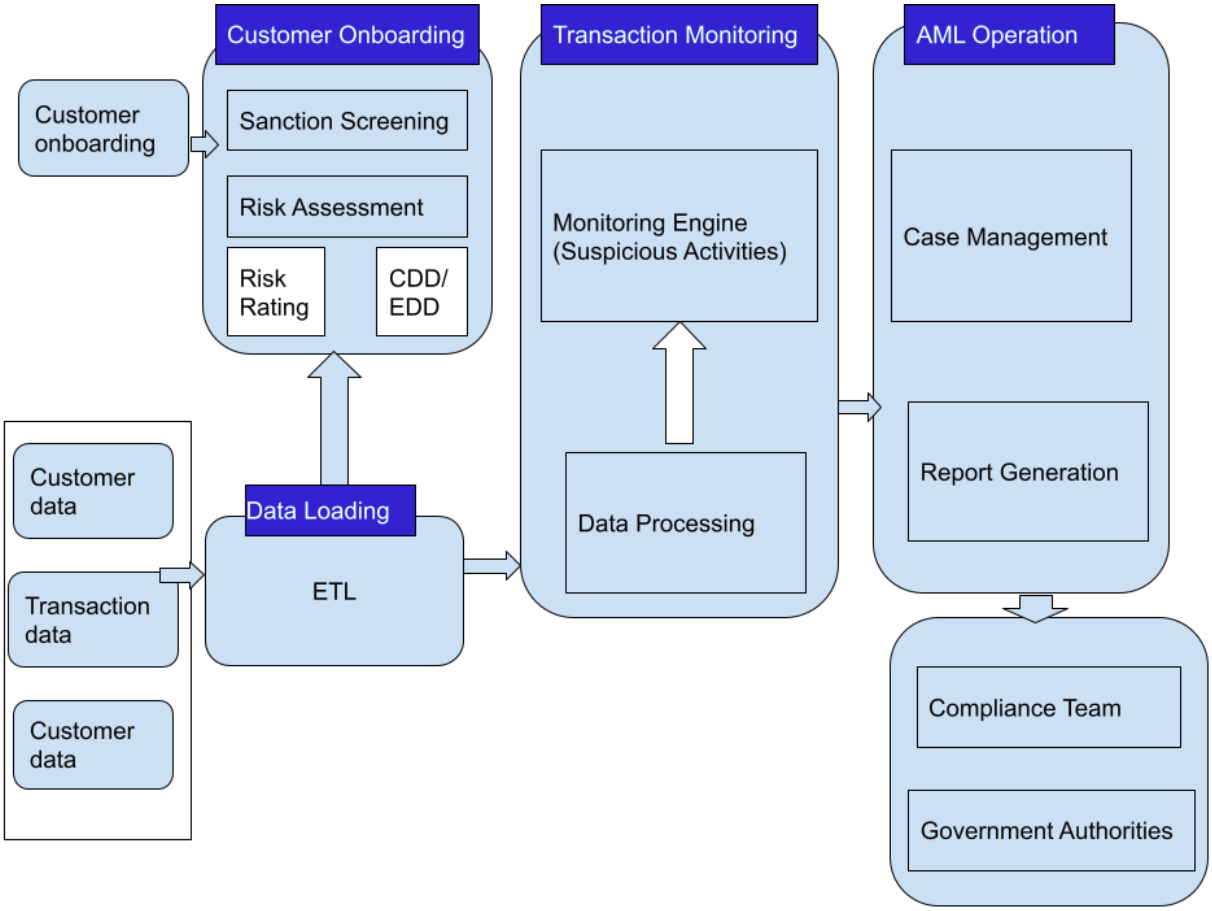

Figure 5. AMT/CTF detection System.

3 Implementation

The implementation section discusses an in-depth explanation of the Match_Recognize Python library contents and functions. The library is designed to allow the users flexible, and secure implementation regardless of the dataset usage. The Match_Recognize Python library consists of a single class containing several functions. A detailed description of the Match_Recognize Python library functions is mentioned below:

3.1 Partition By Clause Function

The *PARTITION BY* clause in Match_Recognize is dedicated to partitioning the input then performing the matching individually for each resulting partition.

A function dedicated for the Match_Recognize *PARTITION BY* clause is created to perform the matching on each specified partition. The function searches for the *PARTITION BY* clause, then returns a list of the partitioned expressions to proceed with the matching.

```
def partition_handling (text):
        start = "PARTITION BY "
        end = "\n"
        grouping = text[text.find(start) + len(start):text.rfind(end) ].split( '\ n' )[0]
        grouping_list = grouping.split(" , " )
        grouping_list = [i.strip() for i in grouping_list]
        return grouping_list
```

3.2 Order By Clause Function

The *ORDERED BY* clause in Match_Recognize is dedicated to ordering the individual rows of each partition before being passed to the Match_Recognize operator. The order_handling function in the Match_Recognize Python library is created to perform the same functionality. It searches the Match_Recognize clause for the *ORDERED BY* clause then separates the expression and returns a list of the ordered by expressions to pass it to the class functions.

```
def order_handling(text):
        start = "ORDER BY "
```

```
    end = "\n"
    ordering = text[ text.find(start) + len(start):text.rfind(end) ].split( '\ n' )[0]
    ordering_list = ordering.split( " , " )
    ordering_list = [i.strip() for i in ordering_list]
    return ordering_list
```

3.3 Measures Clause Function

A function dedicated to the Match_Recognize is created to detect the output specified by the users and return it as the class output. The Measures_handling function searches the Match_Recognize query input for the *MEASURES* clause then captures the user's specified output, separates it as a list of data elements then returns it in order to be used elsewhere in the class.

```
def MEASURES_handling(text):
        start = "MEASURES"
        end = "PATTERN"
        txt = text[ text.find(start) + len(start):text.rfind(end) ]
        txt = txt.replace( " \ n  " ,   " " )
        txt = txt.replace("      " ," " )
        txt = txt.replace("    " ," " )
        txt = txt.replace(" \ t " ," " )
        txt_list = txt.split( " , " )
        return txt_list
```

3.4 Define Clause Detection

The *DEFINE* clause in Match_Recognize specifies the pattern to be matched within the data. The define_handling function searches the Match_Recognize query for the *DEFINE* clause and separates the text to pass it to be used within the library. It is important to mention that this function's aim is to detect the DEFINE clause then pass the pattern logic to be used in the proceeding functions.

```
def define_handling(text):
```

```
    start = "DEFINE"
    end = " ) "
    txt =  text[ text.find(start) + len(start):text.rfind(end) ]
    txt = txt.replace( " \ n  " ,   " " )
    txt = txt.replace("      " ," " )
    txt = txt.replace("    " ," " )
    txt = txt.replace(" \ t " ," " )
    txt_list = txt.split( " , " )
    return txt_list
```

3.5 Define Clause Separation

The Match_Recognize *DEFINE* clause consists of:

1. symbol: pattern variable.
2. as: considered to be the *DEFINE* clause separator from the symbol and the expression.
3. expression: the logic to be matched on the data.

The aim of this function is to separate the building blocks of the *DEFINE* clause to use them as follows:

1. Symbol: to be passed through the class pipeline to be used to identify each logic.
2. as: separates the symbol from expression, hence will not be used within the library.
3. expression: to be passed through the pipeline in order to be used as the pattern logic that corresponds to each symbol.

```
def definition_splitting(define_text):
    dict_ = {}
    for i in define_text:
        key_ = i.split('AS ' )[0].lstrip().value
        _ = i.split('AS ' )[1].lstrip()
        dict_[key_] = value_
    return dict_
```

3.6 Define Expression Separation

This function is dedicated for splitting the expression of the DEFINE clause logic to determine the data attribute specified then return it as the comparison building block.

```
def splitting_define(text):
    before_ = text[text.find('')+len(''):text.rfind(check_operand(text))].replace(" ", "")
    operand_ = check_operand(text)
    after_caluse= text[text.find(check_operand(text))+len(check_operand(text)):text.rfind('')].replace(" ", "")
    after_ = after_caluse[after_caluse.find('(')+len('('):after_caluse.rfind(')')].replace(")", "")
    return before_, operand_, after_
```

3.7 Expression Operand Validation

The check_operand function is responsible for the validation of the operand passed within the *DEFINE* clause expression. The check_operand function ensures the user has passed a valid operand and provides an error message if the user passed a non-valid operand as an arithmetic operand.

```
def check_operand(text):
        final_operand = ''
        if text.find('<') != -1:
                return '<'
        elif text.find('<=') != -1:
                return '<='
        elif text.find('>') != -1:
                return '>'
        elif text.find('>=') != -1:
                return '>='
        elif text.find('=') != -1:
                return '='
        elif text.find('!=') != -1:
                return '!='
        else:
                print('error in check_operand')
```

3.8 Convert Expression Logic

The get_logic function is responsible for converting the *DEFINE* clause logic from a list of string elements into an arithmetic output. The string operands obtained from the *DEFINE* clause separation are converted into an arithmetic representation and implemented on the building blocks of the expression comparison list obtained from the expression separation function.

```
def get_logic(inp, relate, cut):
    ops = {'>': operator.gt,
           '<': operator.lt,
           '>=': operator.ge,
           '<=': operator.le,
           '=': operator.eq,
           '!=': operator.ne}
    return ops[relate](inp, cut)
```

3.9 Previous Window Function Support

The Match_Recognize Python library also supports the implementation of the *prev()* window function. The prev() window function navigates to the previous row to apply the specified expression. The prev_handling function navigates to the previous row and implements the comparison specified within the *DEFINE* expression.

```
def prev_handling(df, column1_, operator_, column2_):
    a = []
    for i, row in df.iterrows():
       if(i != 0):
          a.append(df.loc[i] if get_truth(row[column1_], operator_, df.iloc[i-1][column2_]) else None)
    return pd.DataFrame([i for i in a if i is not None], columns = df.columns.tolist()).reset_index(drop = True)
 def numeric_handling(df, column1_, operator_, value_):
    a = []
    for i, row in df.iterrows():
       if(i != 0):
          a.append(df.loc[i] if get_truth(row[column1_], operator_, int(value_)) else None)
```

```
    return pd.DataFrame([i for i in a if i is not None], columns = df.columns.tolist()).reset_index(drop = True)
```

3.10 Logic Assembly

The assembly_func function is dedicated to applying the logic of the pattern defined in the *PATTERN* clause using its corresponding logic. It loops through each logic to implement the logic on the data then uses each output as an input to the next iteration.

```
def assembly_func(txt, df):
    df_ = df
    dict_ = {}
    text = definition_splitting(define_handling(txt))
    for keys, values in text.items():
     if('prev' in values.lower()):
        column1_, operator_, column2_ = splitting_define(values)
        df1 = prev_handling(df_, column1_, operator_, column2_)
        dict_[keys] = df1
     else:
        column1_, operator_, value_ = values.split(" ")
        df1 = numeric_handling(df_, column1_, operator_, value_)
        dict_[keys] = df1
    return dict_
```

3.11 Expression detection

The expression_detection function detects the differences between the regular expression quantifiers. Currently the Match_Recognize Python library supports the usage of two quantifiers which are the plus sign *+* and the star sign ***. The function searches through the pattern and separates the symbols to apply the quantifiers logic with the corresponding expression logic on the data. The function returns two lists, one containing the symbols with the plus and star quantifiers.

```
def expression_detection(txt):
```

```
    txt[txt.find("PATTERN")+len("PATTERN"):txt.rfind(")")]
    l = (txt.split("PATTERN ("))[1].split(")")[0].strip().split(" ")
    star_list = [i.strip()[:-1] for i in l if '*' in i]
    plus_list = [i.strip()[:-1] for i in l if '+' in i]
    return star_list, plus_list
```

3.12 Match Implementation

The match_final function implements the Match_Recognize query logic on the dataset records. Starting with specifying the symbols within the *MEASURES* clause and specifying the star and plus quantifiers symbols, then use the *PARTITION BY* clause partitioning function in order to split the logic implementation for each partition and check the first and last window functions in the MEASURES clause in order to implement the specified output of the Match_Recognize clause, then Implementing the expressions logic specified in the define clause for each symbol in the quantifier in order to return the output in the same order mentioned in the *MEASURES*, and *PARTITION BY* clauses.

```
def match_final(df,txt):
    firsts_, lasts_, dictionary_ = dictionary_creation(df, txt)
    data_length = min([len(lasts_[keys]) for keys, values in dictionary_.items()])
    col = [val.split(" ")[-1] for val in [i.strip() for i in MEASURES_handling(txt)]]
    star_list, plus_list = expression_detection(txt)
    [val[val.find(" AS ")+len(" AS "):val.rfind("")] for val in MEASURES_handling(txt)]
    partitioning = partition_handling(txt)
    output_ = pd.DataFrame(columns = partitioning+col)
    ordering = order_handling(txt)
    for i in MEASURES_handling(txt):
        val = i
        first_ = pd.DataFrame(columns = df.columns)
        last_ = pd.DataFrame(columns = df.columns)
        for keys, values in dictionary_.items():
            lasts_[keys] = lasts_[keys][:data_length].reset_index(drop = True)
            firsts_[keys] = firsts_[keys][:data_length].reset_index(drop = True)
            values = values.sort_values(by = ordering).reset_index(drop = True)
            if(keys.strip() in plus_list):
                if i.find(keys.strip()) != -1 and val[val.find("")+len(""):val.rfind("(")].lower() == 'last':
```

```
                output_[val[val.find(" AS ")+len(" AS "):val.rfind("")]] =
lasts_[keys][val[val.find(".")+len("."):val.rfind(")")]].tolist()
            if i.find(keys.strip()) != -1 and val[val.find("")+len(""):val.rfind("(")].lower() == 'first':
              output_[partitioning] = firsts_[keys][partitioning].reset_index(drop = True)
              output_[val[val.find(" AS ")+len(" AS "):val.rfind("")]] =
firsts_[keys][val[val.find(".")+len("."):val.rfind(")")]].tolist()
    output_ = output_[output_[col[0]] > output_[col[-1]]]
    output_ = output_[partitioning+col]
    return output_
```

3.13 Quantifier Logic

The match_recogize function validates the input with the quantifier logic to ensure the correctness of the regular expression in comparison to the pattern logic implemented on the data. The function checks both the star and plus quantifiers and scans the data record by record to verify the quantifier corresponding logic to return the dataset after validation. The process is a particularly critical step, especially in complex patterns with multiple quantifiers.

```
def match_recognize(df, txt):
    act_output = match_final(df, txt)
    if(select_handling(txt)[0] != '*'):
      act_output = act_output[select_handling(txt)]
    else:
      act_output = act_output
    return act_output
```

3.14 Match Recognize Automaton

The match_automaton function introduces the Match_Recognize Python library automaton. Starting with converting the user's input regular expression into a set of symbols to be tested using the automaton and specifying the dictionary representing the automaton transition table. The function verifies the Match_Recognize pattern and transforms it into the input symbols used within the automaton to create the set of states list based on the number of symbols, followed by iterating over the set of states

to create the table of transitions for each state. The function then ingests the set of states, input symbols and transition table into a non-deterministic [Bro89] automaton to validate the user input on the automaton and returning a message 'accepted' in the case the user's input was applicable on the automaton or 'rejected' if the input were not valid.

```
def match_automata(txt, txt_):
    dicto = {}
    pattern_ = (txt.split("PATTERN ("))[1].split(")")[0].strip().split(" ")
    pattern_ = [i[:-1] for i in pattern_]
    pattern_rep = [chr(ord('`')+i+1) for i in range(len(pattern_))]
    reps_ = [i[:-1] for i in txt_.split(" ")]
    star_list, plus_list = expression_detection(txt)
    nfa_string = [pattern_rep[i] for i in range(len(pattern_)) if pattern_[i] in reps_]
    star_list_ = [pattern_rep[i] for i in range(len(pattern_)) if pattern_[i] in star_list]
    plus_list_ = [pattern_rep[i] for i in range(len(pattern_)) if pattern_[i] in plus_list]
    set_ = set([ f'q{i}' for i in range(len(set(star_list + plus_list)))])
    qs = list(set_)
    qs.sort()
    subdict = {}
    for i in range(len(qs)):
       if(i != len(qs)-1):
         if(pattern_rep[i+1] in plus_list_):
           dicto[qs[i]] = {pattern_rep[i]: {qs[i]}, pattern_rep[i+1]: {qs[i+1]}}
         else:
           dicto[qs[i]] = {pattern_rep[i]: {qs[i]}, pattern_rep[i+1]: {qs[i+2]}}
       else:
         dicto[qs[i]] = {pattern_rep[i]: {qs[i]}}

    nfa = NFA(
       states=set([ f'q{i}' for i in range(len(set(star_list + plus_list))+1)]),
       input_symbols=set(pattern_rep),
       transitions= dicto
       ,
       initial_state=qs[0],
       final_states={qs[-1]}
    )
    quantifier_validation = [x for x in [i[:-1] for i in txt_.split(" ") if i[-1] == '*'] if x in plus_list]
```

```
if (len(quantifier_validation) == 0) & (pattern_ == reps_) & (nfa.accepts_input(nfa_string)):
    return 'accepted'
else:
    return 'rejected'
```

4 Evaluation

This section is dedicated to designing and developing the rules and use cases using the Match_Recognize library then implementing it in the Oracle database. As well as performing simulations on the Match_Automaton function where a new pattern will be passed to the Match_Automaton function to examine whether a new regular match with the Match_Recognize expression. The Match_Automaton function uses an NFA automaton to test the new pattern passed to the Match_Automaton function as a regular expression on the old pattern specified in the Match_Recognize pattern clause. This is particularly useful since one of the big issues with the financial institution and organization is having many use case scenarios that performs the same monitoring using the same pattern recognition but for various categories e.g., having two use case scenarios to detect frequent transactions for high-risk countries and another for tax havens or having multiple use case scenarios with the same logic to monitor different customer groups. which can affect the use case scenarios negatively since the more use case scenarios injected into the transaction monitoring engine, the more resources it consumes, and it causes overhead on the financial institution and organization servers. The optimal objective of financial institutions and organizations is to have unique, effective, and concise use case scenarios that can cover all ML/TF schemes efficiently. The Match_Automaton function can be used to test new developed patterns using the existing use case scenarios. If specific patterns can be detected using a wider range pattern, then with the help of threshold setting, the duplicated use case scenarios can be excluded e.g., if the same logic for detecting frequent transactions for tax havens use case scenario is accepted on the frequent transactions for high-risk countries use case scenario, then one of the use case scenarios can be excluded and by modifying the use case scenario threshold to include tax heavens and high-risk countries the AML/CTF logic is achieved using fewer and more effective monitoring setting.

4.1 Use Case Scenarios:

This subsection will discuss these use case scenarios in detail and describe what makes them truly a state-of-the-art Fraud, and AML/CTF detection scenarios that can be used in any financial institution and organization.

To perform the validation process, some of the most versatile, crucial regulatory entity rules and use case scenarios have been designed and implemented.

Due to the limited bank transactions dataset, we are using both publicly available and private real time bank transactions.

The dataset shown in the screenshots is the private real time bank transactions which consists of 116822 records and the following columns:

1. Customer ID: Describes customers unique Identification, the dataset consists of 1000 unique customers. Customer ID column is crucial since when financial regulatory entities detects and validates the presence of suspicious activities such as money laundering or terrorist financing schemes, they are required to file a suspicious transaction report (STR) [str16] which requires the inclusion of the customer information which is gathered by the customer identification.

2. Transaction Type: States different transaction types which can be Credit or Debit transaction. The transaction type is important in the detection of suspicious activities since different transaction types are handled differently with different restrictions and monitoring thresholds that will be mentioned below.

3. Transaction Amount: Presents the transaction amount regardless of being credit or debit transaction. Transaction amounts are the values compared with the monitoring threshold to determine if this is a suspicious activity.

4. CounterParty Country: Specifies the transaction recipient country. CounterParty Country is crucial for use case scenarios that monitor high-risk jurisdictions.

5. Timestamp: Refers to the transaction completion timestamp. With the help of transaction timestamps, users can specify time frames to inject to the Match_Recognize library.

Real time bank transactions were used to shed special light on the process of suspicious activity detection and handling. Based on the Financial Intelligence Unit, as soon as a financial institution and organization detects any suspicious activities it is obligated to add the customer's information into a suspicious transaction report (STR) [str16].
The following datasets were used in the validation process:

1. 1999 Czech Financial Dataset [Pet22b] containing 1,056,320 records alongside 10 columns: transaction id, account id, date, transaction type, operation, amount, balance, characterization symbol, bank of the partner, account.

2. Bank Transaction [KSS20] with 116,201 records containing customer's account numbers, date, transaction description, cheque number information, value date, withdrawal amount, deposit amount, balance.

Rules and use case scenarios were designed and developed based on the FATF 40 Recommendation [FAT] for the purpose of implementing their logic using a simple declarative regular expression pattern that replaces the static single-usage use case scenario code.
All use case scenarios are valid for any financial institution and organization whether it's a local, international, online financial institution and organization.
The designed rules and use case scenarios described below:

1. U-shaped Transactions: The most well-known use case scenarios due to its versatility. It can be used with different transaction types: Credit, Deposit, Cash, International, etc.
   Additionally, the U-shape creates a pattern that makes it hard for financial institutions and organizations to properly detect and specify the list of customers performing the suspicious activity pattern. The use case scenario should be divided for each customer identification number. The scenario should observe each customer's transaction amount volume and detect the data stream in which the customer transaction amount volume gets rapidly decreasing in comparison to the previous transaction volumes then quickly increases again.

2. Suspiciously large Transactions: Another critical use case scenario is the suspiciously large transactions scenario which is dedicated to identifying any suspicious customer behavior outside the customer's normal behaviors. To specify the customer's usual behavior from suspicious behavior, financial institutions and organizations should divide their data into several groups, which gives the financial institutions and organizations the ability to specify the normal usual threshold that the transaction amount volume will be compared with for each group to verify whether this is indeed a suspicious activity or not. What makes this use case scenario truly one of the most powerful scenarios is its utter versatility and simplicity. The use case scenario can be used with any transaction type: Credit, Debit, Cash, Cheque and because there are four different transaction types, financial institutions and organizations can have four different use case scenarios in result, since each scenario will be dedicated for the validation of each transaction type. Assuming financial institutions and organizations have limited customer groups: children, teens, adults, wealth, PEP, small business, investors, corporate, students, and public sector. With the help of the use case scenario, financial institutions and organizations can create ten different scenarios by passing different threshold amounts to be compared with each transaction

amount for each customer group.

3. Inconsistent Transactions: An additional powerful use case scenario is the inconsistent transactions scenario. It detects transactions that start small which can be dedicated when the current transaction amount increases in comparison to the previous transaction amount then followed by a period of flatness in transaction amount which is followed by a noticeable decrease in transaction value. The use case scenario can also be implemented on any customer group, transaction type, and with their respected threshold values.

4. Incremental Transactions: A cardinal use case scenario is the incremental transaction scenario where financial institutions and organizations become more empowered to detect transactions just below the limit followed by suspicious customer behavior. With this scenario's help, financial institutions and organizations can benefit from two well-known suspicious behavior patterns. The first suspicious behavior pattern included in the scenario is having credit slips just below the customer group or transaction type threshold so that these transactions will not be monitored on the system and the unusual, suspicious behavior which can be specified by each financial institution and organization.
   The second suspicious behavior pattern is pipelining the output of the credit slips with a gradual increase in transaction amount volumes.

5. High-Risk Jurisdiction Transactions: A foremost use case scenario for all financial institutions and organizations always is the high-risk jurisdiction transactions scenario. Even though the use case scenario logic is straightforward, all transactions sent to high-risk jurisdictions must be monitored using strict grouped thresholds to be able to detect any fraudulent activities as quickly, and swiftly as possible. Using the use case scenario, not only financial institutions and organizations will be able to monitor these transactions but also to specify proper thresholds to restrict any future suspicious activity connected to the high-risk jurisdictions.
6. Overseas Transactions: The final use case scenario is dedicated to monitoring overseas transactions. Overseas transactions are identified as transactions in which the initiation country is not the destination country. Overseas transactions are very salient to be monitored because countries do not have the same Anti-Fraud and AML/CTF guidelines and restrictions. Indicating that the customer may perform suspicious activities with suspicious entities, individuals, businesses

and the financial institution and organization will not be able to find out using normal means.

4.2 Scenarios Implementation on Oracle Match_Recognize

The subsection will present the Oracle database implementation of the Match_Recognize clause on the real time dataset and other bank transaction datasets for each use case scenario mentioned in the previous subsection.

1. U-shaped Transactions: The Anti-Fraud, and AML/CTF use case scenario detects transactions forming U shapes. Which can be identified by transaction volumes starting high then rapidly decreasing followed by quickly increasing back again. The use case scenario can be represented by the following regular expression (DOWN+ UP+) where the DOWN symbol refers to the pattern where the transaction amount volume starts high then begins to decrease, and the UP symbol resembles the pattern when the transaction amount volume starts low then increases gradually over time.

```
SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY
customer_id
 ORDER BY      timestamp
 MEASURES              FIRST (UP.timestamp )       AS       first_tstamp,
                      LAST(DOWN.timestamp )      AS      bottom_tstamp
 PATTERN   (DOWN+             UP+)
 DEFINE
 UP AS transaction_amount> PREV( transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount) )
```

2. Suspiciously large Transactions: This Anti-Fraud, and AML/CTF use case scenario compares each data attribute with the threshold value to distinguish all the above the threshold transactions indicated suspicious activity. Each financial institution can determine which thresholds are considered suspicious and which are considered normal customer behavior. The example shows that for each transaction a threshold of 475 euro is considered normal. On the other hand, if transactions exceed the threshold of 2000 transactions, then the scenario opts to monitor it.

```
SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE ( PARTITION BY
customer_id
 ORDER BY timestamp
 MEASURES FIRST (NORMAL.timestamp ) AS first_tstamp ,
 LAST(UNUSUAL.timestamp ) AS bottom_tstamp
PATTERN (NORMAL+ UNUSUAL+)
DEFINE
 NORMAL AS transaction_amount < 20 ,
UNUSUAL AS transaction_amount > 2300)
```

3. Inconsistent Transactions: The inconsistent transactions Anti-Fraud, and AML/CTF use case scenario detects inconsistencies in customer's transaction behavior. The scenario observes the transactions that are fluctuating in nature which is identified as a small transaction amount that gradually increases until it reaches a point where it remains consistent then starts to decrease again. The corresponding regular expression used within the scenario is represented as the following: (UP+ FLAT+ DOWN+) where the UP symbol represents an increase in transaction amount volume in comparison to the previous transaction amount volumes. The FLAT symbol represents a period of consistency in transaction amount volumes. Identified as the current transaction amount volume is equal to the previous transaction amount volumes, followed by a decrease in transaction amount volume in comparison to the previous transaction amount volumes.

```
SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE ( PARTITION BY
customer_id
 ORDER BY timestamp
 MEASURES FIRST (UP.timestamp) AS first_tstamp,
 LAST(DOWN.timestamp) AS bottom_tstamp
PATTERN (UP+ FLAT+ DOWN+)
DEFINE
UP AS transaction_amount > PREV( transaction_amount) ,
FLAT AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount))
```

4. Incremental Transactions: This scenario pipelines one of the most recognized Money Laundry and Terrorist Finance transaction criminal behaviors which is transactions just below the thresholds. Criminals perform transactions just below the threshold to prevent the monitoring engine from detecting it. This scenario is catered for this monitoring system gap as it will gather all the below the threshold transactions to filter out the below the threshold transactions followed by an increase in transaction amount volumes.

```
SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE ( PARTITION BY
customer_id
 ORDER BY timestamp
 MEASURES FIRST (BELOW.timestamp ) AS first_tstamp ,
 LAST(UP.timestamp ) AS bottom_tstamp
PATTERN (BELOW+ UP+)
DEFINE
 BELOW AS transaction_amount < 20 ,
UP AS transaction_amount>PREV(transaction_amount) )
```

5. High-risk Jurisdiction Transactions: Monitoring Transactions linked to high-risk jurisdiction is critically important due to the fact that high-risk jurisdiction are identified as jurisdictions with weak or poor Anti-Fraud, and AML/CTF controls resulting in a risk of dealing with transactions of money laundry and terrorist financing nature in the financial institution and organization from one end and not considering it to be of a suspicious from the high-risk jurisdiction's financial institution and organization end. For each identified high-risk jurisdiction, financial institutions and organizations must specify thresholds to prevent transaction amount volumes from exceeding the respected threshold. Simultaneously, the same logic can be applied to tax havens.

   Meaning that this scenario can be used not only for high-risk jurisdiction but tax havens as well using the same concept and implementation. The scenario detects transaction amount volumes linked to high-risk jurisdiction and tax havens exceeding the threshold followed by a period of consistency then decrease in transaction amount volumes. The scenario regular expression pattern can be represented as the following: (ABOVE+ FLAT* DOWN+) where the ABOVE symbol indicates transaction amount volumes exceeding the threshold, followed by the FLAT symbols representing a period of consistency in transaction amount

volumes, and the DOWN symbol indicating a decrease of transaction amount volumes.

```
SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE (
 PARTITION BY customer_id , counter_party_country
 ORDER BY timestamp
 MEASURES FIRST (ABOVE.timestamp ) AS first_tstamp,
 LAST(DOWN.timestamp ) AS bottom_tstamp
PATTERN (ABOVE+ FLAT* DOWN+)
DEFINE
 ABOVE AS transaction_amount > 500 ,
 FLAT AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount)) t
where counter_party_country = 'RO '
```

6. Overseas Transactions: This is a fundamental Anti-Fraud, and AML/CTF scenario for monitoring overseas transactions to detect fraudulent activities when doing transactions where the country of origin is not the destination country. The scenario logic is similar to the high-risk jurisdiction scenario logic however the main difference is that the destination country specified in the overseas transactions' scenario is not a high-risk nor a tax haven jurisdiction, it is an overseas country which is different from the originating payment country. The pattern monitored by the scenario can be identified as the following: (ABOVE+ UP+ FLAT* DOWN+) for all transaction amount volumes above the specified threshold followed by an inconsistent activity similar to the inconsistent transactions' scenario, however the consistency period specified by the symbol FLAT is not detrimental for this specific scenario. The scenario observes transactions related to overseas jurisdictions starting small then gradually increasing followed by a decrease. The scenario also filters out transactions that have a consistent period between the increase and decrease periods.

```
SELECT * FROM
(SELECT transaction_table1.* FROM transaction_table1
    inner join transaction_table2
```

```
on transaction_table1.customer_id=transaction_table2.customer_id and
    transaction_table1.counter_party_country!=transaction_table2.counter_party_country)
 MATCH_RECOGNIZE (
   PARTITION BY customer_id, counter_party_country
   ORDER BY timestamp
   MEASURES FIRST (ABOVE.timestamp ) AS first_tstamp ,
            LAST(DOWN.timestamp ) AS bottom_tstamp
PATTERN (ABOVE+ UP+ FLAT* DOWN+)
DEFINE
 ABOVE AS transaction_amount > 500 ,
UP AS transaction_amount > PREV(transaction_amount) ,
FLAT AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount) )
```

4.3 Scenarios Implementation using Match_Recognize Library on Python 3.9

This subsection presents the scenario simulations using the Match_Recognize Python library, and explains the library usage, and functionality.

Currently the Match_Recognize Python library runs on the latest Python version 3.9; however, it was tested on earlier versions such as Python 3.9, 3.8 and 3.7.

Users need to initiate a new instance of the class by calling the class name first "match_recognize()" and store its value in a class variable. This way allows the users to call any function within the Match_Recognize Python library. The Match_Recognize Python library has a function called "match_recognize" that takes two arguments: The dataset on which users perform the pattern recognition, and the Match_Recognize clause query.

Below is an example on how to create an instance of the Match_Recognize Python library and use it to call the "match_recognize" function then store the output of the "match_recognize" function in a variable called "match_output" for further use:

```
class_var = match_recognize()
match_output = class_var.match_recognize(dataset, match_recognize query)
```

Simulations of the Anti-Fraud, and AML/CTF use case scenarios are presented below, each query implemented in Oracle database is passed as the Match_Recognize clause query to the Match_Recognize function in the Match_Recognize Python library and the output is stored in the variables presented below:

1. U-shaped Transactions: For the U-shaped Transactions scenario, users need to create an instance of the Match_Recognize library by initiating a new class object "match_recognize()" then use the stored class object to call the "match_recognize" function and pass it the dataset alongside the Match_Recognize U-shaped query.

```
txt = " " "SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY customer_id
    ORDER BY     timestamp
    MEASURES            FIRST (UP.timestamp)      AS      first_tstamp ,
                        LAST(DOWN.timestamp)     AS     bottom_tstamp
    PATTERN  (DOWN+           UP+)
    DEFINE
    UP   AS      transaction_amount > PREV(transaction_amount) ,
    DOWN AS   transaction_amount < PREV(transaction_amount))
" " "
match_recognize_class = match_recognize()
u_shape = match_recognize_class.match_recognize(data_set,txt)
```

Alongside the scenario simulation, the scenario pattern was validated as well on the Match_Recognition Python library Automaton function Match_Automaton in Fig. 6. The proposed pattern compared with the original Match_Recognize pattern was (UP* DOWN+) which indicates zero or more occurrences in the UP symbol which is different from the one or more occurrences the UP symbol supports in the Match_Recognize Python library. The result of the library run was successfully rejecting the pattern.

```
txt = """SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE(
      PARTITION BY customer_id
      ORDER BY timestamp
      MEASURES    FIRST(UP.timestamp) AS first_tstamp,
                  LAST(DOWN.timestamp) AS bottom_tstamp
      PATTERN (UP+ DOWN+)
      DEFINE
      UP AS transaction_amount > PREV(transaction_amount),
      DOWN AS transaction_amount < PREV(transaction_amount)
      )
"""
mach_recognize_class = match_recognize()
u_shape = mach_recognize_class.match_automata(txt, "UP* DOWN+")
u_shape
```

```
'rejected'
```

Figure 6. U-shaped scenario pattern validation on Match_Automaton.

2. Suspiciously large Transactions: After simulating the Suspiciously large Transactions scenario using the Match_Recognize library, a new class object is created followed by calling the "match_recognize" function to pass the Match_Recognize clause query as an input to the "match_recognize" function alongside the dataset.

```
txt = " " "SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY customer_id
ORDER BY     timestamp
 MEASURES        FIRST (NORMAL.timestamp)      AS      first_tstamp ,
                 LAST(UNUSUAL.timestamp)       AS     bottom_tstamp
PATTERN      (NORMAL+         UNUSUAL+)
DEFINE
 NORMAL AS    transaction_amount < 20 ,
  UNUSUAL AS  transaction_amount > 2300)
" " "
match_recognize_class = match_recognize()
suspiciously_large = match_recognize_class.match_recognize(data_set,txt)
```

The Automaton validation check for the Suspiciously large Transactions scenario was conducted using the Match_Automaton function, presented in Fig. 7 in which the proposed pattern was introduced (NORMAL+ UNUSUAL*) which matches with

zero or more occurrences for the UNUSUAL regular expression to which the Match_Automaton function correctly rejects.

```
txt = """SELECT * FROM transactions MATCH_RECOGNIZE (
    PARTITION BY customer_id
    ORDER BY timestamp
    MEASURES  FIRST(NORMAL.timestamp) AS first_tstamp,
              LAST(UNUSUAL.timestamp) AS bottom_tstamp
    PATTERN (NORMAL+ UNUSUAL+)
    DEFINE
    NORMAL AS transaction_amount < 20,
    UNUSUAL AS transaction_amount > 2300
    )
"""
mach_recognize_class = match_recognize()
suspiciously_large = mach_recognize_class.match_automata(txt, "NORMAL+ UNUSUAL*")
suspiciously_large
```

```
'rejected'
```

Figure 7. Suspiciously large Transactions scenario pattern validation on Match_Automaton.

3. Inconsistent Transactions: To properly simulate the inconsistent transaction scenario users need to initiate a new Match_Recognize Python library class object to use it to call the "match_recognize" function then pass it the Match_Recognize scenario clause with the dataset.

```
txt = " " " SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY
customer_id
ORDER BY     timestamp
 MEASURES         FIRST (UP.timestamp)       AS       first_tstamp,
                  LAST(DOWN.timestamp)    AS     bottom_tstamp
PATTERN      (UP+      FLAT+     DOWN+)
DEFINE
 UP AS transaction_amount > PREV(transaction_amount) ,
FLAT AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount)  )
 " " "
 match_recognize_class = match_recognize()
```

inconsistent_transactions = match_recognize_class.match_recognize(data_set,txt)

For the Inconsistent Transactions use case scenario, the Match_Automaton function validation check presented in Fig. 8 specifies that the proposed regular expression validation pattern (UP+ FLAT* DOWN+) which matches with zero or more occurrences for the FLAT symbol. However, in the Match_Recognize pattern, it is observed that the FLAT symbol matches with one or more occurrences indicating that at least one occurrence should be achieved. Therefore, the Match_Automaton function rejected the new pattern.

```
txt = """select * from transactions MATCH_RECOGNIZE (
    PARTITION BY customer_id
    ORDER BY timestamp
    MEASURES  FIRST(UP.timestamp) AS first_tstamp,
              LAST(DOWN.timestamp) AS bottom_tstamp
    PATTERN (UP+ FLAT+ DOWN+)
    DEFINE
    UP AS transaction_amount > PREV(transaction_amount),
    FLAT AS transaction_amount = PREV(transaction_amount),
    DOWN AS transaction_amount < PREV(transaction_amount)
    )
"""
mach_recognize_class = match_recognize()
inconsistent_transactions = mach_recognize_class.match_automata(txt, "UP+ FLAT* DOWN+")
inconsistent_transactions
```

```
'rejected'
```

Figure 8. Inconsistent Transactions scenario pattern validation on Match_Automaton.

4. Incremental Transactions: Similar to previous examples, to properly simulate the Incremental Transactions use case scenario, users need to initiate a new Match_Recognize Python library class object to use it in calling the "match_recognize" function to be able to pass the Match_Recognize Incremental Transactions scenario clause alongside the dataset.

```
txt = " " "SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY
    customer_id
ORDER BY    timestamp
  MEASURES      FIRST (BELOW.timestamp)     AS     first_tstamp ,
```

```
                    LAST(UP.timestamp)      AS      bottom_tstamp
PATTERN      (BELOW+          UP+)
DEFINE
 BELOW AS    transaction_amount < 20 ,
 UP  AS transaction_amount> PREV(transaction_amount) )
" " "
match_recognize_class = match_recognize()
incremental_transactions = match_recognize_class.match_recognize(data_set,txt)
```

Since the Incremental Transactions scenario pattern uses two regular expressions. The Match_Automaton function validation check presented in Fig. 9 checks whether the NFA automaton will accept the occurrence of a new regular expression symbol within the Match_Recognize specified regular expression symbols indicating the occurrence of a new matching. To which the Match_Automaton library has successfully rejected.

```
txt = """SELECT distinct customer_id FROM transactions MATCH_RECOGNIZE (
     PARTITION BY customer_id
     ORDER BY timestamp
     MEASURES  FIRST(BELOW.timestamp) AS first_tstamp,
               LAST(UP.timestamp) AS bottom_tstamp
    PATTERN (BELOW+ UP+)
    DEFINE
    BELOW AS transaction_amount < 20,
    UP AS transaction_amount > PREV(transaction_amount)
    )
"""
match_recognize_class = match_recognize()
incremental_transactions = match_recognize_class.match_automata(txt, "BELOW+ FLAT* UP+")
incremental_transactions
```

```
'rejected'
```

Figure 9. Incremental Transactions scenario pattern validation on Match_Automaton.

5. High-risk Jurisdiction Transactions: Since the High-risk Jurisdiction use case scenario is conducted for specified high-risk jurisdictions that dynamically change overtime and different for each financial institution and organization. It is important not to use specific static jurisdiction lists while performing the simulation of the scenario. For this particular use case scenario, it is crucial to ingest the high-risk jurisdictions list then run the use case scenario on these transactions.

```
txt = " " "SELECT * FROM TRANSACTIONS MATCH_RECOGNIZE( PARTITION BY customer_id
ORDER BY     timestamp
  MEASURES        FIRST (ABOVE.timestamp)       AS       first_tstamp,
                  LAST(DOWN.timestamp)      AS      bottom_tstamp
PATTERN      (ABOVE+          FLAT*     DOWN+)
DEFINE
 ABOVE AS   transaction_amount > 500 ,
 FLAT    AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount) )
" " "
match_recognize_class = match_recognize()
high_risk_jurisdiction = match_recognize_class.match_recognize(data_set,txt)
```

Regarding the High-risk Jurisdiction Transactions use case scenario Automaton validation test. The Match_Recognize pattern clause for the use case scenario uses three regular expressions which the validation test tries to delete the occurrence of the first regular expression symbol identified as the ABOVE symbol indicating that the start state of the new pattern is different from the start state of the Match_Recognize function, resulting in the Match_Automaton function successfully rejecting the pattern in Fig. 10.

```
txt = """SELECT * FROM transactions MATCH_RECOGNIZE (
     PARTITION BY customer_id, counter_party_country
     ORDER BY timestamp
     MEASURES  FIRST(ABOVE.timestamp) AS first_tstamp,
               LAST(DOWN.timestamp) AS bottom_tstamp
    PATTERN (ABOVE+ FLAT* DOWN+)
    DEFINE
    ABOVE AS transaction_amount > 500,
    FLAT AS transaction_amount = PREV(transaction_amount),
    DOWN AS transaction_amount < PREV(transaction_amount))
    """
match_recognize_class = match_recognize()
high_risk_jurisdiction = match_recognize_class.match_automata(txt, "FLAT* DOWN+")
high_risk_jurisdiction

'rejected'
```

Figure 10. High-risk Jurisdiction Transactions scenario pattern validation on Match_Automaton.

6. Overseas Transactions: The overseas Transactions use case scenario monitors suspicious activities where the transaction-initiated country is not the destination country. Because for each jurisdiction the overseas jurisdiction list is different, it is important that the dataset ingested in the use case scenario simulation is for the overseas transactions only.

```
txt = " " "SELECT * FROM TRANSACTIONS
 MATCH_RECOGNIZE    (
   PARTITION    BY customer_id , counter_party_country
   ORDER BY     timestamp
   MEASURES         FIRST (ABOVE.timestamp)       AS       first_tstamp ,
                    LAST(DOWN.timestamp)      AS      bottom_tstamp
PATTERN (ABOVE+      UP+       FLAT*     DOWN+)
DEFINE
 ABOVE AS   transaction_amount > 500 ,
UP   AS transaction_amount > PREV(transaction_amount) ,
 FLAT    AS transaction_amount = PREV(transaction_amount) ,
DOWN AS transaction_amount < PREV(transaction_amount) )
" " "
match_recognize_class = match_recognize()
overseas_transactions = match_recognize_class.match_recognize(data_set,txt)
```

The Overseas Transactions use case scenario has the highest count of symbols in its pattern regular expression, resulting in the Match_Automaton function validation check presented in Fig. 11 to test whether the NFA automaton accepts the deletion of the last symbol of the pattern regular expression or rejects it as the deletion indicates that the new pattern will not reach the final state specified in the Match_Recognize clause to which the NFA automaton has successfully rejected.

```
txt = """SELECT * FROM
    transactions
    MATCH_RECOGNIZE (
     PARTITION BY customer_id, counter_party_country
     ORDER BY timestamp
     MEASURES  FIRST(ABOVE.timestamp) AS first_tstamp,
               LAST(DOWN.timestamp) AS bottom_tstamp
    PATTERN (ABOVE+ UP+ FLAT* DOWN+)
    DEFINE
    ABOVE AS transaction_amount > 500,
    UP AS transaction_amount > PREV(transaction_amount),
    FLAT AS transaction_amount = PREV(transaction_amount),
    DOWN AS transaction_amount < PREV(transaction_amount)
    )
"""
match_recognize_class = match_recognize()
overseas_transactions = match_recognize_class.match_automata(txt, "ABOVE+ UP+ FLAT*")
overseas_transactions
```

```
'rejected'
```

Figure 11. Overseas Transactions scenario pattern validation on Match_Automaton.

4.4 Comparison

This section discusses the comparison results of simulating the use case scenarios using the Match_Recognize Python library and static simulations of the use case scenarios using Python. As well as comparing the results between the Match_Recognize clause Oracle simulation and Match_Recognize Python library simulation for each use case scenario. Since the Match_Recognize clause is not supported in financial institutions, organizations, The data analysts, and scientists need to simulate these scenarios statically for each jurisdiction because as observed above, each jurisdiction has dedicated thresholds. The data analysts and scientists will need to study each use case scenario and recreate it manually from scratch. This process is very time consuming to the extent that it takes up to 9 hours for an experienced expert to simulate an individual use case scenario for a single jurisdiction. The experiment compares the simulation of the six use case scenarios using the Match_Recognize Python library and manually recreating, running, and finalizing the simulation by a senior data analyst. A record of the number of hours it took the data analyst to simulate the six-use case scenario mentioned in the use case scenarios subsection for a total of four jurisdictions is shown below:

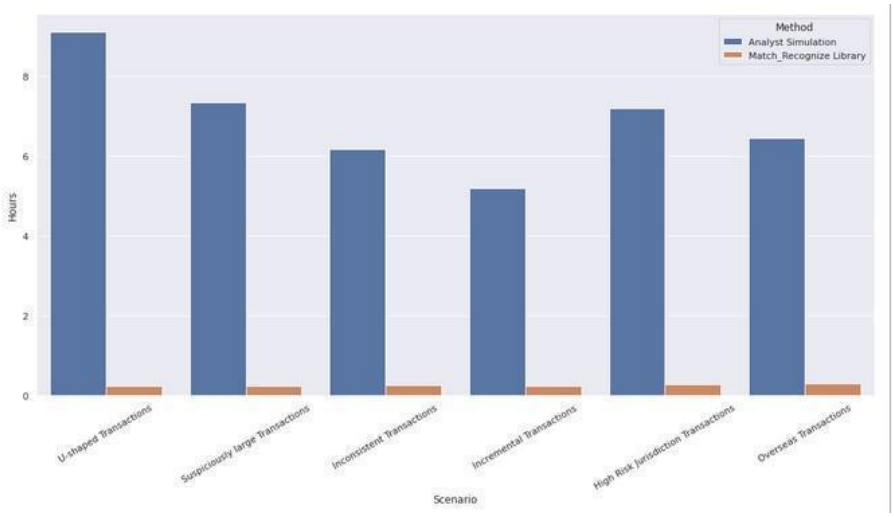

Figure 12. AMT/CTF detection System.

As observed above in Fig. 12 that using the Match_Recognize Python library for simulating the use case scenarios instead of statically simulating each scenario is proven to save almost 96.3% of the time data analysts or data scientists require to recreate, run, and finalize the simulations. Moreover, the Match_Recognize Python library users do not need to consume the institution and organization resources due to the fact that using the Python library, users only need to create an instance of the library then pass the Match_Recognize clause to it rather than creating the simulation using static Python code that requires memory allocation for the code on a regular basis.

The table below 2 presets the number of unique customers that matched with each Anti-Fraud, and AML/CTF use case scenario pattern:

Scenario Name	Oracle Simulation	Match_Recognize Simulation
U-shaped Transactions	9885	9885
Suspiciously large Transactions	68	68
Inconsistent Transactions	18	18
Incremental Transactions	93	93

High-risk Jurisdiction Transactions	72	69
Overseas Transactions	9955	9941

Table 2. Difference between Oracle and Python Library Simulation.

The above table shows the number of customers with transactions that indeed matched with the pattern specified within the Match_Recognize Python library. There are some inconsistencies in the number of customers matched with the pattern, which is related to the long pattern. Therefore, it is recommended that for the current version of the library 0.0.3 keep the regular expression pattern symbols under four unique symbols.

5 Limitations

The limitations section lists the current version of the Match_Recognize Python library limitations:

1. Negation: Negation using the ∧ symbol matching with 0 occurrences in the regular expression of the pattern sub-clause is not supported yet in the pattern handling function of the Match_Recognize Python library. Match_Recognize pattern sub-clause specify negation in the regular expression such as ’UP+ FLAT* ∧DOWN’.

2. Limited Pattern Quantifiers Handling: The Match_Recognize Python library only supports the two most used quantifiers, the "+" quantifier which matches with one more occurrence, and the "*" quantifier which matches with zero or more occurrences.

3. Limited Parsing: The Match_Recognize clause performs semantic checks to recognize the meaning of the query. The Match_Recognize Python library does not support semantic checks yet.

4. Limited Error Handling: The Match_Recognize Python library performs limited error handling in the form of static error messages specifying the root cause of the error with limited support for the ways to solve the error. More error handling functions will be added in the next versions of the library to give users more insights into practical solutions.

6 Conclusion

The goal of the thesis is to introduce an optimal approach for replacing the statically specified financial fraud detection rules and use case scenarios specified by the financial regulatory entities with a declarative solution that eliminates the need for designing and developing dedicated, single use scenarios that are incorporated into the transaction monitoring engine as an additional layer for detecting fraudulent activities for each financial fraud use case scenario specified by the financial regulatory entities.

The introduced Match_Recognize Python library eliminates the need for designing and implementing static rules and use case scenarios and instead uses simple regular expressions to detect fraudulent activities.

The Match_Recognize library consists of multiple functions, presented in the implementation section. The match_recognize function in the Match_Recognize Python library functions takes a regular Match_Recognize clause and the dataset to return the data matched with the specified pattern. The background section describes the Match_Recognize clause in detail.

Additionally, the Match_Recognize Python library automaton utilizes the Match_Recognize data stream to dynamically generate a non-deterministic automaton and validate new patterns using it which is presented in section three.

The thesis provides designed and developed six of the most recognized Anti-Fraud, AML/CTF use case scenarios that are built based on the FATF 40 Recommendation that mimic real ML/TF schemes and facilitate the simulation process of both Oracle and Match_Recognize Python library of the Match_Recognize Python library which is discussed in detail in the evaluation section.

The evaluation section also compares the Match_Recognize Oracle database simulation results and the Match_Recognize Python Library simulation results to compare the time consumed by each approach.

The comparison between the time required for conducting simulations for each scenario using the static approach versus performing the simulations using the Match_Recognize Python library has shown a reduction in the time required for each scenario simulation to be successfully finished of approximately 96.3% which concludes that the purpose of the thesis was successfully achieved, especially when the Match_Recognize Python library didn’t take as much of memory allocation as commonly used within the static simulations. Finally, the Match_Recognize Python library usage is not limited to only financial fraud detection applications, it can also be used in a wider range of applications such as medical, and commercial applications.

References

[FRDA] S. E., Mohga, and A., Ahmed. Financial Fraud Detection: A Declarative Approach. (2023).

[AGAG21] Amber Qin Ant Group and Sophia Wu Ant Group. What future aml compliance requires: A technology perspective, 2021.

[AS17] A Acharya and NS Sidnal. Framework to process high frequency trading using complex event processing. *International Journal of Knowledge Based Computer System*, 5(1):17–25, 2017.

[BH20] Kim Berg Hansen. Up-and-down patterns. In *Practical Oracle SQL*, pages 325–349. Springer, 2020.

[Bro89] J Glenn Brookshear. *Theory of computation: formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[DFKRJ12] Jack W Dorminey, Arron Scott Fleming, Mary-Jo Kranacher, and Richard A Riley Jr. Financial fraud. *The CPA Journal*, 82(6):61, 2012.

[BTC06] Benjamin C Brodie, David E Taylor, and Ron K Cytron. A scalable architecture for high-throughput regular-expression pattern matching. ACM SIGARCH computer architecture news, 34(2):191–202, 2006.

[FIN] FATF, Terrorist Financing Risk Assessment Guidance, FATF, Paris, www.fatf-gafi.org/publications//methodsandtrends/documents/Terrorist-Financing-Risk-Assessment- Guidance.html, (2019).

[FAT] FATF, International Standards on Combating Money Laundering and the Financing of Terrorism & Proliferation, FATF, Paris, France, www.fatf-gafi.org/recommendations.htm. (2012-2021).

[CSD] Kharisma, Dona Budi, and Afilya Hunaifa. "Comparative study of disgorgement and disgorgement fund regulations in Indonesia, the USA and the UK." Journal of Financial Crime, (2022).

[Ful] Matt Fuller, Rss feed complex event detection. [Online] Available: https://cs.brown.edu/research/pubs/theses/masters/2008/fuller.pdf, 2008.

[Han] Matt Fuller Kim Berg Hansen, Practical oracle sql. [Online] Available: https://link.springer.com/book/10.1007/978-1-4842-5617-6, 2020.

[HP03] Haruo Hosoya and Benjamin C Pierce. Regular expression pattern matching for xml. *Journal of Functional Programming*, 13(6):961–1004, 2003.

[ISO] Iso/iec tr 19075-5:2016 information technology—database languages—sql technical reports—part 5: Row pattern recognition in sq.

[JDB] Das, Tulika, Venkatasubramaniam Iyer, Elizabeth Hanes Perry, Brian Wright, Thomas Pfaeffle, Brian Martin, Ashok Shivarudraiah Irudayaraj et al. "Oracle Database JDBC Developer's Guide, 12c Release 1 (12.1) E49300-05.", 2014.

[Kör21] Michael Körber. Accelerating event stream processing in on-and offline systems. 2021.

[mat22] Weik, M.H., pattern-matching. In: Computer Science and Communications Dictionary. Springer, Boston, MA. https://doi.org/10.1007/1-4020-0613-6_13732, 2000.

[mon] FATF, FATF President's paper: Anti-money laundering and counter terrorist financing for judges and prosecutors, FATF, Paris, France, www.fatf-gafi.org/publications/methodandtrends/documents/AML-CFT-judges-prosecutors.html, 2018.

[rec22] L.A., McNeely, C.L, Pattern Recognition. In: Schintler, L.A., McNeely, C.L. (eds) Encyclopedia of Big Data. Springer, Cham. https://doi.org/10.1007/978-3-319-32010-6_300166, 2022.

[reg22] Jabotinsky, H.Y. (2019). Financial Regulation. In: Marciano, A., Ramello, G.B. (eds) Encyclopedia of Law and Economics. Springer, New York, NY. https://doi.org/10.1007/978-1-4614-7753-2_636, 2019.

[str16] BELIZE, Money Laundering & Terrorism (Prevention) Act. [Online] Available: https://fiubelize.org/wp-content/uploads/2016/06/MLTPA-as-Amended-2016.pdf, 2008.

[sus] BELIZE, Types of Suspicious Activities or Transactions. [Online] Available: https://fiubelize.org/types-of-suspicious-activities-or-transactions. 2016.

[KSS20] A Kullaya Swamy and B Sarojamma. Bank transaction data modeling by optimized hybrid machine learning merged with arima. *Journal of Management Analytics*, 7(4):624–648, 2020.

[Lak17] Keith Laker. Deep dive into 12c match_recognize. pages 1–177, 2017.

[Nah19] Mohammed Ahmad Naheem. Anti-money laundering/trade-based money laundering risk assessment strategies–action or re-action focused? *Journal of Money Laundering Control*, 2019.

[Pet22a] Dušan Petkovic. Specification of row pattern recognition in the sql standard´ and its implementations. *Datenbank-Spektrum*, pages 1–12, 2022.

[Pet22b] Liz Petrocelli, 1999 czech financial dataset - real anonymized transactions [dataset], 2022.

[Rep18] Alex Reprintsev. Row pattern matching: match_recognize. In *Oracle SQL Revealed*, pages 199–216. Springer, 2018.

[RMP17] Maxim Repin, Oleg Mikhalsky, and Ekaterina Pshehotskaya. Architecture of transaction monitoring system of central banks. In *International Conference" Actual Issues of Mechanical Engineering" 2017 (AIME 2017)*, pages 654–658. Atlantis Press, 2017.

[ZSF17] Mary Frances Zeager, Aksheetha Sridhar, Nathan Fogal, Stephen Adams, Donald E Brown, and Peter A Beling. Adversarial learning in credit card fraud detection. In *2017 Systems and Information Engineering Design Symposium (SIEDS)*, pages 112–116. IEEE, 2017.

Appendix

I. Glossary

Money Laundry	The process of concealing the origin of money, often obtained from illicit activities.
Anti-Money Laundering	The execution of transactions to eventually convert illegally obtained money into legal money.
Terrorist Finance	The solicitation, collection or provision of funds with the intention that they may be used to support terrorist acts or organizations.
Counter Terrorist Financing	A framework that seeks to stop the flow of illegal cash to terrorist organizations. It is closely tied to anti-money laundering (AML).
High-risk Jurisdiction.	Countries and Authorities with weak measures to combat money laundering and terrorist financing (AML/CFT).
Financial Intelligence Unit	Domestic units that act as a central gathering point to receive and analyse suspicious transaction reports [str16].
Typology	The pattern and series of events that lead to a financial crime.
STR	A report indicating customer's suspicious activities which is filed to the Financial Intelligence Unit.
Pattern	The sequence of two tokens.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Mohga Soliman Emam Hewashy,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Financial Fraud Detection: A Declarative Approach,
   *(*title of thesis)

   supervised by Ahmed Awad.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mohga Soliman Emam Hewashy
05/01/2023