Integration of Databricks with CRDP

Databricks [Overview]

This document describes how to configure and integrate CipherTrust Manager with Databricks.

Built on open source and open standards, a lakehouse simplifies your data estate by eliminating the silos that historically complicate data and AI. One architecture for integration, storage, processing, governance, sharing, analytics and AI. One approach to how you work with structured and unstructured data. One end-to-end view of data lineage and provenance. One toolset for Python and SQL, notebooks and IDEs, batch and streaming, and all major cloud providers. Thales provides a few different capabilities to protect data in Databricks. The most secure option is to Bring Your Own Encryption (BYOE). There are a couple of ways to utilize this capability with Databricks.

Bring Your Own Encryption (BYOE)

- **Data Ingest** – with Thales Batch Data Transformation (BDT) or User Defined Functions (UDF)
- **Data Access** –user defined functions for column level protection.

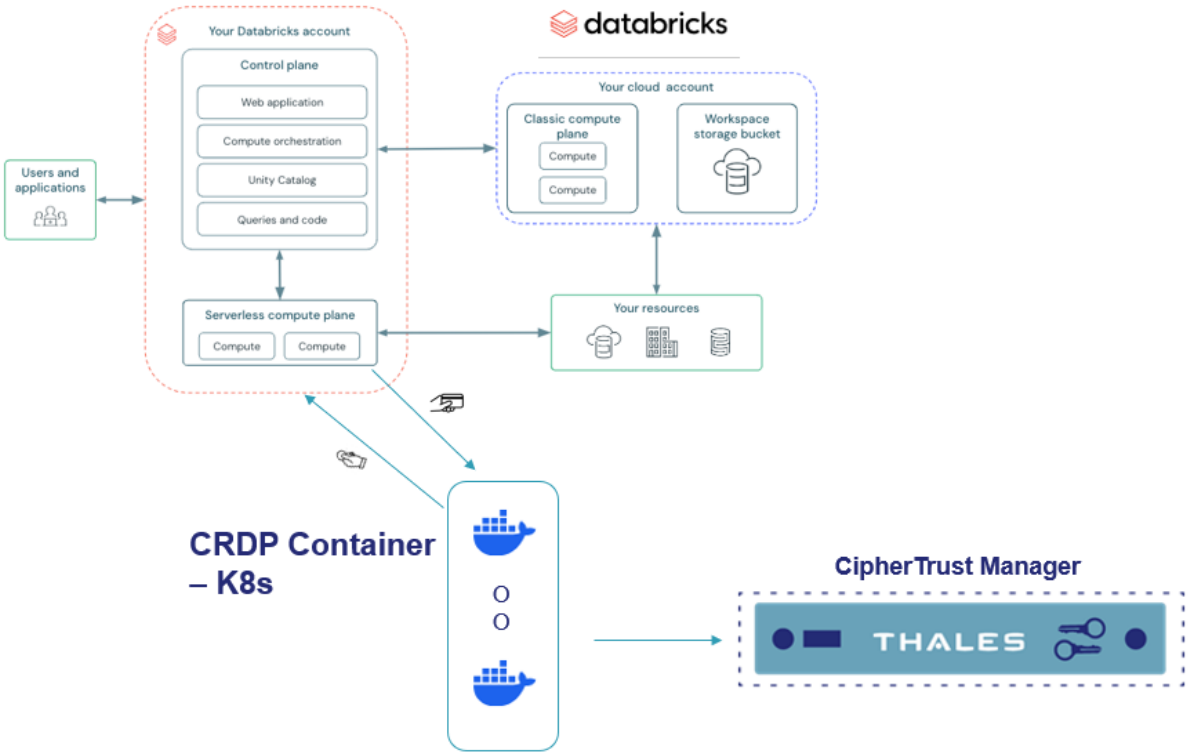
The focus of this integration will be on Data Access protecting sensitive data in Databricks columns by using CipherTrust REST Data Protection(CRDP)

There are both Java and Python examples provided.

Architecture

Databricks can run on all three major cloud service providers AWS, Azure and GCP. A nice feature with Databricks is that even though each CSP has its own unique way to create functions and gateways there is only one method to know with Databricks. Listed below is an example of how this integration works.

Databricks Sample Integration



Supported Product Versions

- **CipherTrust Manager** CipherTrust Manager 2.14 and higher.
- **CRDP** CRDP 1.0 was tested.
- **Databricks Compute LTS 14.3 or higher.**

This integration is validated using Java 1.8. As of October 16 higher version of Java were not supported by Databricks.

Prerequisites

- Helpful Databricks links:
  - https://docs.databricks.com/en/udf/unity-catalog.html
- Ensure that CRDP container is installed and configured. Refer to https://thalesdocs.com/ctp/con/crdp/latest/admin/crdp-deploy_alternative/index.html
- Ensure that the CipherTrust Manager is installed and configured. Refer to the **CipherTrust Manager documentation** for details.

Java UDF's are currently only supported in Databricks Compute Cluster. Although you can execute the Java UDF's in a notebook to execute against a table in Databricks SQL Warehouse

they are not supported in Databricks SQL Warehouse as a standalone UDF. Here is an example of what is supported from a Databricks notebook:

```
%sql
select ThalesencryptCharUDF(c_name) as enc_name, c_name from samples.tpch.customer
limit 50
```

Steps for Integration Java Example

- **[Installing and Configuring Thales CRDP container]**
- **[Download code from Thales github and compile]**
- **[Publish jar file to Databricks Compute Engine]**
- **[Define and register the Java UDF in Databricks Notepad]**
- **[Integration with Thales CipherTrust Manager]**
- **[UDF Environment or property file Variables]**

Installing and Configuring CRDP Container

To install and configure **CDRP,** refer to Quick Start.

Eclipse development tool was used for these examples. Here is the version used for testing along with the Maven plugin for Eclipse.

Eclipse development tool was used for these examples. Here is the version used for testing along with the Maven plugin for Eclipse.

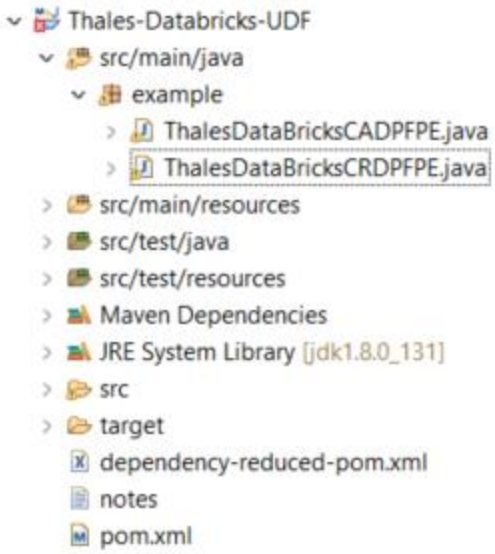
eclipse.buildId=4.15.0.I20200305-0155

m2e - Maven Integration for Eclipse

Download code from github and compile.

git clone https://github.com/ThalesGroup/CipherTrust_Application_Protection.git

The database directory has all the code for Databricks.



Assuming you have your CM already configured each of the classes have a main method which can be used to test locally. Make sure your CM environment is configured correctly. This example uses a configuration file (udfConfig.properties) in the java project resource directory to set variables needed by the UDF. Environment variables or a secrets manager can also be used for this information. (See section on Environment Variables for details).

Generate the jar file to upload to Databricks. (Java)

To compile and generate the target jar file to be uploaded to the Databricks compute platform select the project and choose “Run As” “maven install” to generate the target.

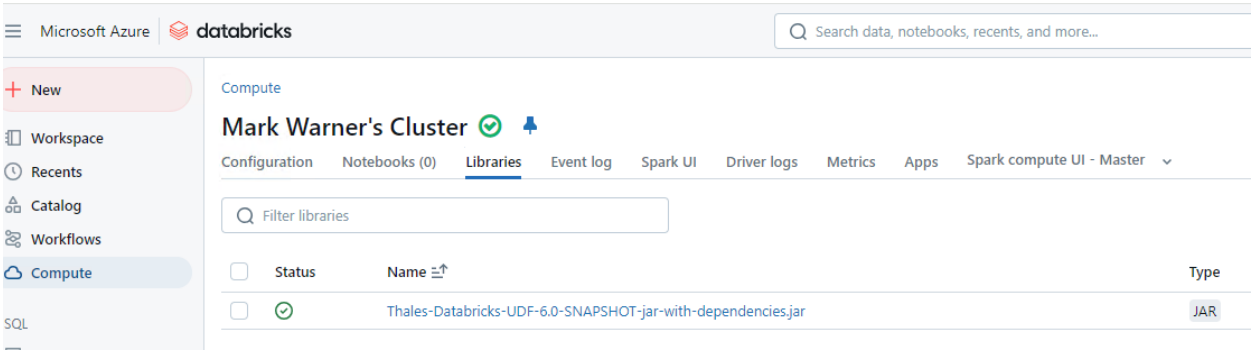
```
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ Thales-Databricks-UDF
---
[INFO] Installing C:\Users\t0185905\workspace\Thales-Databricks-UDF\target\Thales-
Databricks-UDF-6.0-SNAPSHOT.jar to C:\Users\t0185905\.m2\repository\Thales\Thales-
Databricks-UDF\6.0-SNAPSHOT\Thales-Databricks-UDF-6.0-SNAPSHOT.jar
[INFO] Installing C:\Users\t0185905\workspace\Thales-Databricks-UDF\pom.xml to
C:\Users\t0185905\.m2\repository\Thales\Thales-Databricks-UDF\6.0-SNAPSHOT\Thales-
Databricks-UDF-6.0-SNAPSHOT.pom
[INFO] Installing C:\Users\t0185905\workspace\Thales-Databricks-UDF\target\Thales-
Databricks-UDF-6.0-SNAPSHOT-jar-with-dependencies.jar to
C:\Users\t0185905\.m2\repository\Thales\Thales-Databricks-UDF\6.0-SNAPSHOT\Thales-
Databricks-UDF-6.0-SNAPSHOT-jar-with-dependencies.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.482 s
[INFO] Finished at: 2024-10-17T10:28:59-04:00

[INFO] ------------------------------------------------------------------------
```

Publish jar/zip file to Databricks compute engine.

Once you have generated the jar file to upload you can then upload it to the Databricks compute engine. Here are the steps:

- Go to the "Workspace" section.
- Click on "Create" -> "Library".
- Upload your JAR file.

Here is the screen that shows the jar file uploaded:



Once you have uploaded the functions above and if you have already configured and setup CM with the key and all the environment variables you can proceed with the next section.

Define and register the Java UDF in Databricks Notepad.

A nice feature with Databricks is that even though each CSP has its own unique attributes to create functions and gateways there is only one method to know with Databricks. Listed below is a summary of the entire process for Java UDF's.

1. Set Up Your Environment

You will need a Databricks compute cluster of 14.3 or higher. Note as of October 2020 Java 1.8 was supported by Databricks.

2. Create the Thales Function in Java

Should already be done from previous steps.

3. Package the Java Code as a JAR

Should already be done from previous steps.

4. Upload the JAR to Databricks

Should already be done from previous steps. You can do this through the Databricks UI:

5. Register the UDF in Databricks

As you can see from below these functions accept 3 input values.

Key	sampledata	Description
datatype	char	datatype of column (char or nbr)
mode	protect	mode of operation(protect,or reveal
Data	345-23-3345	Can be protected or sensitive information.

Register Examples in Databricks Notebook

Character Data

```
%scala
import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.expressions.UserDefinedFunction

val ThalesprotectCharUDF: UserDefinedFunction = udf((data: String) => {
  try {
    example.ThalesDataBricksCRDPFPE.thales_crdp_udf(data,"protect","char")
  } catch {
    case e: Exception => null
  }
})

spark.udf.register("ThalesprotectCharUDF", ThalesprotectCharUDF)
spark.sql("CREATE or replace FUNCTION ThalesprotectCharUDF AS
'example.ThalesprotectCharUDF'")
spark.sql("SELECT ThalesprotectCharUDF('thisisatest')").show()
```

```
%scala
import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.expressions.UserDefinedFunction

val ThalesrevealCharUDF: UserDefinedFunction = udf((data: String) => {
  try {
    example.ThalesDataBricksCRDPFPE.thales_crdp_udf(data,"reveal","char")
  } catch {
    case e: Exception => null
  }
})

spark.udf.register("ThalesrevealCharUDF", ThalesrevealtCharUDF)
```

```
spark.sql("CREATE or replace FUNCTION ThalesrevealtCharUDF AS
'example.ThalesrevealCharUDF'")
spark.sql("SELECT ThalesrevealCharUDF('3eJQrKCLCJH')").show()
```

Number Data

```
%scala
import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.expressions.UserDefinedFunction
val ThalesprotectNbrUDF: UserDefinedFunction = udf((data: String) => {
  try {
    example.ThalesDataBricksCRDPFPE.thales_crdp_udf(data,"protect","nbr")
  } catch {
    case e: Exception => null
  }
})

spark.udf.register("ThalesprotectNbrUDF", ThalesprotectNbrUDF)
spark.sql("CREATE or replace FUNCTION ThalesprotectNbrUDF AS
'example.ThalesprotectNbrUDF'")

spark.sql("SELECT ThalesprotectNbrUDF('3453453345345')").show()
```

```
%scala
import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.expressions.UserDefinedFunction

val ThalesrevealNbrUDF: UserDefinedFunction = udf((data: String) => {
  try {
    example.ThalesDataBricksCRDPFPE.thales_crdp_udf(data,"reveal","nbr")
  } catch {
    case e: Exception => null
  }
})

spark.udf.register("ThalesrevealNbrUDF", ThalesrevealNbrUDF)
spark.sql("CREATE or replace FUNCTION ThalesrevealNbrUDF AS
'example.ThalesrevealNbrUDF'")
//spark.sql("SELECT ThalesrevealNbrUDF('10020001460941821452')").show()
spark.sql("SELECT ThalesrevealNbrUDF('233682980')").show()
```

Integration with CipherTrust Manager.

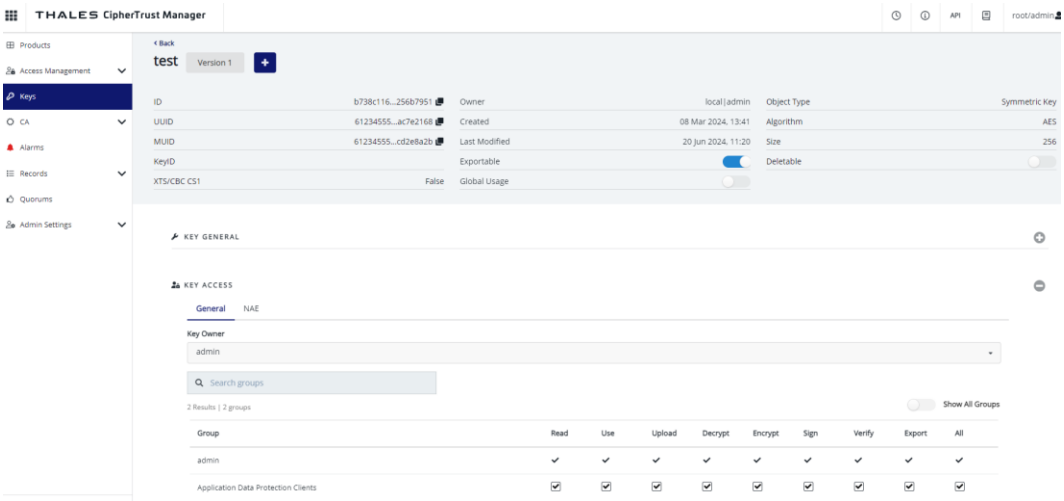
Here is a link to a demo that shows all the steps required to setup CRDP in CM.

thales.navattic.com/thalesprotectreveal

Here is a list of the steps.

1. Create an Encryption Key
2. Create a CRDP Application
3. Create a User Set
4. Create an Access Policy
5. Create a Protection Policy
6. Perform Integration

When creating a key to be used by the protection policy make sure that it is in the Application Data Protection Clients Group.



When all the above steps are performed you should be able to execute a query within a session that has already defined the function. Here is a sample query running in a databricks notepad cell using one of the UDF's. Please note this must run after the registration above in the same notebook.

```
%sql
select ThalesprotectNbrUDF(c_custkey) as enc_c_custkey, c_custkey from samples.tpch.customer limit 5
```

(1) Spark Jobs

Table

	enc_c_custkey	c_custkey
1	1002000128091	412445
2	1002000692999	412446
3	1002000290966	412447
4	1002000956658	412448
5	1002000993470	412449

Note: The above example was using an internal policy which includes the CRDP metadata with the protected data.

When using a CRDP policy that is based on an external policy your results would be something like the following:

```
%sql
select ThalesprotectCharUDF(c_name) as enc_name, c_name from samples.tpch.customer limit 50
```

(1) Spark Jobs

Table

	enc_name	c_name
1	INIGNUmQ#k6ooUjm...	Customer#0004124...
2	KAE09OFh#eiQhgLgfl	Customer#0004124...
3	RsqD5rr9#B4spSHZvD	Customer#0004124...
4	HXgul0J5#TVXNJgtS5	Customer#0004124...
5	MIt7SMGA#iuikyynnl	Customer#0004124...
6	2sCOUmly#DWFqWIc1r	Customer#0004124...
7	JimlvvsT#hMzaKHH6o	Customer#0004124...
8	FMjtvV8N#73GkCX0y8	Customer#0004124...
9	6VZCJsPW#v2h7pmyjW	Customer#0004124...

Python Example

The process to run python code is much simpler than java. All you need to do is create your python application in a Databricks notepad and run it. There are two examples provided. The first uses the CRDP REST API to make calls for each data element and the second will process all

data elements at once using the CRDP REST Bulk API. All examples take in 3 parameters, data values, mode of operation (protect,reveal for the first example), (protectbulk,revealbulk) for the second example and the third parameter is the datatype (char or nbr).

Example 1. (protect,reveal)

```
import requests
import json
import decimal
import pandas as pd
from pyspark.sql.functions import pandas_udf
from pyspark.sql.types import StringType
from pyspark.sql.functions import pandas_udf, lit, udf
from pyspark.sql.types import StringType
from decimal import Decimal, InvalidOperation

# Load properties from a configuration file
properties = {}

with open("udfConfig.properties") as prop_file:
    for line in prop_file:
        name, value = line.partition("=")[::2]
        properties[name.strip()] = value.strip()

REVEALRETURNTAG = "data"
PROTECTRETURNTAG = "protected_data"

def thales_crdp_python_function(databricks_inputdata, mode, datatype):
    encdata = ""

    # Check for invalid data
    if datatype.lower() != "char":
        databricks_inputdata = str(databricks_inputdata)

    # Check for invalid data
    if databricks_inputdata is not None and databricks_inputdata.strip():
        # Check if the length of the input is less than 2
        if len(databricks_inputdata) < 2:
            return databricks_inputdata

        # Check if datatype is not 'char'
        if datatype.lower() != "char":
            lower_bound = -9
```

```
        upper_bound = -1

        try:
            # Convert the string to an integer
            number = Decimal(databricks_inputdata)

            # Check if the number is between -1 and -9
            if lower_bound <= number <= upper_bound:
                print("The input is a negative number between -1 and -9.")
                return databricks_inputdata

        except ValueError:
            print("The input is not a valid number.")
            return databricks_inputdata

#else:
    # Return input if it is None or empty
    #return databricks_inputdata

# Fetch properties
crdpip = properties.get("CRDPIP")
if not crdpip:
    raise ValueError("No CRDPIP found for UDF.")

return_ciphertext_for_user_without_key_access = (
    properties.get("returnciphertextforuserwithnokeyaccess", "no").lower() == "yes"
)
user_set_lookup = properties.get("usersetlookup", "no").lower() == "yes"
key_metadata_location = properties.get("keymetadatalocation")
external_version_from_ext_source = properties.get("keymetadata")
protection_profile = properties.get("protection_profile")

#Print protection profile and key metadata location for debugging
#print("Protection Profile:", protection_profile)
#print("Key Metadata Location:", key_metadata_location)

data_key = "data"
if mode == "reveal":
    data_key = "protected_data"

try:
    json_tag_for_protect_reveal = (
        PROTECTRETURNTAG if mode == "protect" else REVEALRETURNTAG
```

```
        )
        show_reveal_key = (
            properties.get("showrevealinternalkey", "yes").lower() == "yes"
        )

        sensitive = databricks_inputdata

        # Prepare payload for the protect/reveal request
        crdp_payload = {
            "protection_policy_name": protection_profile,
            data_key: sensitive,
        }

        if mode == "reveal":
            crdp_payload["username"] = "admin"
            if key_metadata_location.lower() == "external":
                crdp_payload["external_version"] = external_version_from_ext_source

        # Construct URL and make the HTTP request
        url_str = f"http://{crdpip}:8090/v1/{mode}"
        headers = {"Content-Type": "application/json"}

        response = requests.post(
            url_str, headers=headers, data=json.dumps(crdp_payload)
        )
        response_json = response.json()

        if response.ok:
            protected_data = response_json.get(json_tag_for_protect_reveal)
            if (
                mode == "protect"
                and key_metadata_location.lower() == "internal"
                and not show_reveal_key
            ):
                protected_data = (
                    protected_data[7:] if len(protected_data) > 7 else protected_data
                )
            encdata = protected_data
        else:
            raise ValueError(f"Request failed with status code: {response.status_code}")
    except Exception as e:
        print(f"Exception occurred: {e}")
        if return_ciphertext_for_user_without_key_access:
```

```
            pass
        else:
            raise e

    return encdata

# Register the UDF
thales_crdp_python_udf = udf(thales_crdp_python_function, StringType())
spark.udf.register("thales_crdp_python_udf", thales_crdp_python_udf)

#Test

#Simple test
sensitive = "45444555"
singlevalue = thales_crdp_python_function(sensitive, "protect", "nbr")
print(f"sensitive: {sensitive}, Encrypted: {singlevalue}")

# Step 1: Fetch data using SQL query
query_result_bal = spark.sql(
    "SELECT c_acctbal FROM samples.tpch.customer where c_acctbal > 1000 LIMIT 5"
)
query_result = spark.sql("SELECT c_name FROM samples.tpch.customer LIMIT 5")

# Step 2: Extract names into a list
names = [row.c_name for row in query_result.collect()]
bals = [row.c_acctbal for row in query_result_bal.collect()]

mode="protect"
key_metadata_location = properties.get("keymetadatalocation")
datatype = "nbr"

if mode == "protect":
    if datatype == "char":
        encrypted_names = [thales_crdp_python_function(name, "protect", "char") for name in
names]
        for original, encrypted in zip(names, encrypted_names):
            print(f"Original: {original}, Encrypted: {encrypted}")
        # Apply the UDF to the DataFrame
        df_with_encrypted_names = query_result.withColumn("encrypted_name",
thales_crdp_python_udf(query_result.c_name, lit("protect"), lit("char")))
        # Show the results
        df_with_encrypted_names.show()
    else:
```

```
        encrypted_bals = [thales_crdp_python_function(bal, "protect", "nbr") for bal in bals]
        for original, encrypted in zip(bals, encrypted_bals):
            print(f"Original: {original}, Encrypted: {encrypted}")
else:
    if datatype == "char":
        if (key_metadata_location == "external"):
            reveal_data = ['INlGNUmQ#k6ooUjm2A', 'KAE09OFh#eiQhgLgfI', 'RsqD5rr9#B4spSHZvD',
'HXguI0J5#TVXNJgtS5', 'MIt7SMGA#iuikyynnl']
        else:
            reveal_data = ['1001000hhIQGjma#fH5yPPOda', '1001000rXhadmXl#XxRlFW0YU',
'1001000BvDFNg8p#NbyCunqpT', '1001000w5U1O9iu#59Ulv2yA8', '1001000CzyGv0KB#mFf9pcKUY']
    else:
        if (key_metadata_location == "external"):
            reveal_data = ['7080.44', '7382.75', '1971.05', '8044.16', '2361.92']
            #reveal_data = ['41', '-53', '343075883524', '22262837758', '208755']
        else:
            reveal_data = ['10020007080.44', '10020007382.75', '10020001971.05',
'10020008044.16', '10020002361.92']

    # Call the function with the test data
    reveal_results = [thales_crdp_python_function(protecteddata, mode, datatype) for
protecteddata in reveal_data]
    print("Results:", reveal_results)
```

Sample output

```
Original: Customer#000412445, Encrypted: INlGNUmQ#k6ooUjm2A
Original: Customer#000412446, Encrypted: KAE09OFh#eiQhgLgfI
Original: Customer#000412447, Encrypted: RsqD5rr9#B4spSHZvD
Original: Customer#000412448, Encrypted: HXguI0J5#TVXNJgtS5
Original: Customer#000412449, Encrypted: MIt7SMGA#iuikyynnl
Original: 5358.33, Encrypted: s3Lc.We
Original: 9441.59, Encrypted: bbx9.bw
Original: 7868.75, Encrypted: JYUW.ws
Original: 6060.98, Encrypted: DStY.ZQ
Original: 4973.84, Encrypted: abwo.vE
```

You can also register the function as well:

```
    # Register the UDF
thales_crdp_python_udf = udf(thales_crdp_python_function, StringType())
spark.udf.register("thales_crdp_python_udf", thales_crdp_python_udf)

# Test the function
# Create a DataFrame
df = spark.sql("SELECT c_name FROM samples.tpch.customer LIMIT 50")
```

```
# Apply the UDF to the DataFrame
df_with_encrypted_names = df.withColumn("encrypted_name",
thales_crdp_python_udf(df.c_name, lit("protect"), lit("char")))

# Show the results
df_with_encrypted_names.show()
```

And then reference it in other cells.

```
%scala
spark.sql("SELECT thales_crdp_python_udf(c_name, 'protect', 'char') AS encrypted_name
FROM samples.tpch.customer LIMIT 50").show()
```

```
%sql
SELECT thales_crdp_python_udf(c_name, 'protect', 'char') AS encrypted_name FROM
samples.tpch.customer LIMIT 50
```

Example 2. (protectbulk,revealbulk)

This example will make one call passing in the entire column to be processed.

```
%python
import requests
import json
from decimal import Decimal, InvalidOperation

# Load properties from a configuration file
properties = {}

with open("udfConfig.properties") as prop_file:
    for line in prop_file:
        name, value = line.partition("=")[::2]
        properties[name.strip()] = value.strip()

def check_valid(databricks_inputdata, datatype):
    encdata = ""
    BADDATATAG = "99999999999"
    # Check for invalid data
    if databricks_inputdata is not None and databricks_inputdata.strip():
        # Check if the length of the input is less than 2
```

```
        if len(databricks_inputdata) < 2:
            return BADDATATAG  + databricks_inputdata
        print
        # Check if datatype is not 'char'
        if datatype.lower() != "char":
            lower_bound = -9
            upper_bound = -1

            try:
                # Convert the string to an Decimal
                number = Decimal(databricks_inputdata)

                # Check if the number is between -1 and -9
                if lower_bound <= number <= upper_bound:
                    #print("The input is a negative number between -1 and -9.")
                    return BADDATATAG

            except ValueError:
               #print("The input is not a valid number.")
                return BADDATATAG
    else:
        # Return input if it is None or empty
        return BADDATATAG

    return databricks_inputdata

def prepare_reveal_input(protected_data, protection_policy_name, key_metadata_location,
external_version=None):
    # Base reveal payload structure
    reveal_payload = {
        "protection_policy_name": protection_policy_name,
        "username": "admin",
        "protected_data_array": []
    }

    # Add external version if key_metadata_location is 'external'
    if key_metadata_location == "external" and external_version:
        reveal_payload["protected_data_array"] = [
            {"protected_data": data, "external_version": external_version} for data in
protected_data
        ]
    else:
        reveal_payload["protected_data_array"] = [
```

```
            {"protected_data": data} for data in protected_data
        ]

    return reveal_payload


def thales_crdp_python_function_bulk(databricks_inputdata, mode, datatype):
    encdata = []

    # Convert input data to string if datatype is not 'char'
    if datatype.lower() != "char":
        databricks_inputdata = [check_valid(str(data),datatype) for data in databricks_inputdata]
    print("databricks_inputdata", databricks_inputdata)
    # Fetch properties
    crdpip = properties.get("CRDPIP")
    if not crdpip:
        raise ValueError("No CRDPIP found for UDF.")

    return_ciphertext_for_user_without_key_access = (
        properties.get("returnciphertextforuserwithnokeyaccess", "no").lower() == "yes"
    )
    key_metadata_location = properties.get("keymetadatalocation")
    external_version_from_ext_source = properties.get("keymetadata")
    protection_profile = properties.get("protection_profile")

    if mode == "protectbulk":
        input_data_key_array = "data_array"
        output_data_key_array = "protected_data_array"
        output_element_key = "protected_data"
    else:
        input_data_key_array = "protected_data_array"
        output_data_key_array = "data_array"
        output_element_key = "data"

    print("mode:", mode)
    try:

        show_reveal_key = (
            properties.get("showrevealinternalkey", "yes").lower() == "yes"
        )

        # Prepare payload for bulk protect/reveal request
        if mode == "protectbulk":
```

```
            crdp_payload = {"protection_policy_name": protection_profile,input_data_key_array:
databricks_inputdata}
        else:
            if key_metadata_location == "external":
                crdp_payload = prepare_reveal_input(databricks_inputdata,
protection_profile,key_metadata_location,external_version_from_ext_source)
            else:
                crdp_payload = prepare_reveal_input(databricks_inputdata,
protection_profile,key_metadata_location)

        if mode == "revealbulk":
            crdp_payload["username"] = "admin"

        # Construct URL and make the HTTP request
        url_str = f"http://{crdpip}:8090/v1/{mode}"
        headers = {"Content-Type": "application/json"}

        print("Sending request to URL:", url_str)

        data=json.dumps(crdp_payload)
        print("Sending data:", data)
        response = requests.post(
            url_str, headers=headers, data=json.dumps(crdp_payload)
        )
        response_json = response.json()

        if response.ok:
            protected_data_array = response_json.get(output_data_key_array, [])
            encdata = [item[output_element_key] for item in protected_data_array]
            # Handle response for bulk data
            if mode == "protectbulk" and key_metadata_location.lower() == "internal" and not
show_reveal_key:
                encdata = [
                    data[7:] if len(data) > 7 else data for data in encdata
                ]
        else:
            raise ValueError(f"Request failed with status code: {response.status_code}")
    except Exception as e:
        print(f"Exception occurred: {e}")
        if return_ciphertext_for_user_without_key_access:
            pass
        else:
            raise e
```

```
    return encdata

# Step 1: Fetch data using SQL query
query_result_bal = spark.sql(
    "SELECT c_acctbal FROM samples.tpch.customer where c_acctbal > 1000 LIMIT 5"
)
query_result = spark.sql("SELECT c_name FROM samples.tpch.customer LIMIT 5")

# Step 2: Extract names into a list
names = [row.c_name for row in query_result.collect()]
bals = [row.c_acctbal for row in query_result_bal.collect()]

mode="protectbulk"
key_metadata_location = properties.get("keymetadatalocation")
datatype = "nbr"
# Step 3: Make the call depending on what the mode and datatype is.  Note you must also
#make sure the udfConfig.property file contains settings that are compatible for the test you
#are running.

if mode == "protectbulk":
    if datatype == "char":
        encrypted_names = thales_crdp_python_function_bulk(names, mode, datatype)
        print("Encrypted Names:", encrypted_names)
    else:
        encrypted_bals = thales_crdp_python_function_bulk(bals, mode, datatype)
        print("Encrypted Balance nbr:", encrypted_bals)
else:
    if datatype == "char":
        if (key_metadata_location == "external"):
            reveal_data = ['INlGNUmQ#k6ooUjm2A', 'KAE09OFh#eiQhgLgfI', 'RsqD5rr9#B4spSHZvD',
'HXguI0J5#TVXNJgtS5', 'MIt7SMGA#iuikyynnl']
        else:
            reveal_data = ['1001000hhIQGjma#fH5yPPOda', '1001000rXhadmXl#XxRlFW0YU',
'1001000BvDFNg8p#NbyCunqpT', '1001000w5U1O9iu#59Ulv2yA8', '1001000CzyGv0KB#mFf9pcKUY']
    else:
        if (key_metadata_location == "external"):
            reveal_data = ['7080.44', '7382.75', '1971.05', '8044.16', '2361.92']
            #reveal_data = ['41', '-53', '343075883524', '22262837758', '208755']
        else:
            reveal_data = ['10020007080.44', '10020007382.75', '10020001971.05',
'10020008044.16', '10020002361.92']
```

```
# Call the bulk function with the test data
encrypted_test_data = thales_crdp_python_function_bulk(reveal_data, mode, datatype)
print("Results:", encrypted_test_data)
```

Sample udfConfig.property file:

```
# udfConfig.properties
CMUSER=apiuser
CMPWD=yorupwde!
returnciphertextforuserwithnokeyaccess=yes
CRDPIP=yourcrdpip
keymetadatalocation=external
keymetadata=1001000
protection_profile=alpha-external
showrevealinternalkey=yes
BATCHSIZE=20
CRDPUSER=admin
```

Environment Variables

Listed below are the environment variables currently required for the Cloud Function.

Key	Value	Description
BATCHSIZE	200	Nbr of rows to chunk when using batch mode
CRDPIP	yourip	CRDP Container IP if using CRDP
keymetadata	1001000	policy and key version if using CRDP
keymetadatalocation	external	location of metadata if using CRDP (internal,external)
protection_profile	plain-nbr-ext	protection profile in CM for CRDP
returnciphertextforuserwithnokeyaccess	yes	if user in CM not exist should UDF error out or retur ciphertext
showrevealinternalkey	yes	show keymetadata when issuing a protect call (CRDP)

Advanced Topics

Databricks also publishes a best practice/performance recommendation link that can also provide some options to improve performance.

https://learn.microsoft.com/en-us/azure/databricks/udf/

https://docs.databricks.com/en/sql/language-manual/sql-ref-functions-udf-scalar.html#language-java

For more information, please refer to the Databricks documentation.

Options for keyname

Currently the sample code uses a hard coded key name. Other options to be considered are:

1.)keyname as an environment variable.
2.)keyname as a Databricks secret.
3.)keyname passed in an attribute to the function. Example with a database view.

```
create view employee as
select first_name, last_name,
thales_CRDP_aws_decrypt_char('testfaas', email) as  email
from emp_basic
```

Controlling Access to UDF

In Databricks, Java UDFs typically run within the Spark executor context, which doesn't directly expose the user information. With CRDP 1.1 user information will be provided in the JWT.

Grants

With Databricks it is possible to manage permissions on UDFs (user-defined functions) in Databricks using **Unity Catalog**. Unity Catalog allows you to control access to UDFs (as well as databases, tables, views, etc.) via GRANT and REVOKE commands. This means you can specify which users or roles are allowed to execute certain UDFs by assigning appropriate privileges.

Steps to Grant Permissions on UDFs in Unity Catalog

1. **Create a UDF in Unity Catalog**: When you create a UDF in a catalog that is managed by Unity Catalog, the UDF becomes an object in the catalog. This allows you to manage access to it.

   Example of creating a UDF in SQL:

   ```
   CREATE FUNCTION catalog_name.schema_name.my_udf AS (
       x INT
   ) RETURNS INT
   COMMENT 'UDF to multiply by 2'
   LANGUAGE PYTHON
   RETURN x * 2;
   ```

2. **Grant Execute Permission**: After creating the UDF, you can grant `EXECUTE` permission on it to specific users, groups, or roles.

   Example SQL to grant execute permission to a specific user:

   ```
   GRANT EXECUTE ON FUNCTION catalog_name.schema_name.my_udf TO
   `user@example.com`;
   ```

If you want to grant permissions to a group or role, you would do something like this:

```
GRANT EXECUTE ON FUNCTION catalog_name.schema_name.my_udf TO
`data_scientist_group`;
```

3. **Revoke Execute Permission**: Similarly, you can revoke permissions if needed:

```
REVOKE EXECUTE ON FUNCTION catalog_name.schema_name.my_udf FROM
`user@example.com`;
```

4. **Check Permissions**: To verify the permissions on the UDF, you can use the `SHOW GRANTS` command:

```
SHOW GRANTS ON FUNCTION catalog_name.schema_name.my_udf;
```

Privileges You Can Grant on UDFs

- **EXECUTE**: Allows the user to execute the UDF.
- **USAGE**: Allows the user to reference the schema or catalog containing the UDF. This might be required alongside the `EXECUTE` privilege, depending on your environment setup.

Unity Catalog Requirement

This type of fine-grained access control for UDFs is only available if you are using **Unity Catalog** in Databricks. If you're not using Unity Catalog, you won't be able to manage access control for UDFs directly using GRANT/REVOKE commands. In non-Unity Catalog environments, UDFs generally do not have per-user permission management beyond the standard Databricks workspace access control (e.g., cluster or notebook permissions).

Without Unity Catalog

If you are not using Unity Catalog, a workaround for controlling access to UDFs would be to create separate UDFs or notebooks for different user groups and control access via workspace permissions (like notebook permissions or cluster permissions), but this is not as granular as the control Unity Catalog provides.

Summary

- **Unity Catalog** allows for fine-grained access control on UDFs with `GRANT EXECUTE` and `REVOKE EXECUTE` commands.
- Without Unity Catalog, you'll need to control access through more general workspace or cluster permissions.

Environment or session variables

It is possible to set environment variables or session-level variables in Databricks that are accessible to a UDF, and you can do this without allowing users to modify the values directly. Here's how you can manage this:

1. Using Spark Configuration Properties:

You can set key-value pairs in the **Spark session configuration** which can be accessed by your UDF. This prevents users from directly modifying these variables.

- **Set the variable in the notebook**: You can set these values at the session level using `spark.conf.set`. For example:

  ```
  spark.conf.set("myapp.secretKey", "super_secret_value")
  ```

- **Access the variable in your UDF**: In your UDF, you can access this configuration property using the `SparkContext` or `SparkSession`.

  Example in Java UDF:

  ```
  import org.apache.spark.sql.api.java.UDF1;
  import org.apache.spark.sql.SparkSession;

  public class MyUDF implements UDF1<String, String> {
      @Override
      public String call(String input) throws Exception {
          SparkSession spark = SparkSession.builder().getOrCreate();
          String secretKey = spark.conf().get("myapp.secretKey");
          // Use the secretKey in your logic
          return "Processed with secret: " + secretKey;
      }
  }
  ```

- **Ensure users cannot modify**: Users of the notebook will not be able to modify the `spark.conf` settings unless they have admin-level access to the notebook. If you manage cluster configurations or the session's scope carefully, they won't have direct access to `spark.conf`.

2. Using Databricks Secrets:

Databricks provides **Databricks Secrets**, which are a secure way to store sensitive information (like API keys, database credentials, etc.) and are not visible to the end user. You can configure environment-specific secrets and access them within UDFs without exposing the actual values.

- **Set up the secret**: In Databricks, you can store secrets in a secrets scope using the UI or the CLI:

  ```
  bash
  databricks secrets create-scope --scope myapp_secrets
  databricks secrets put --scope myapp_secrets --key secretKey
  ```

- **Access the secret in your UDF**: You can access the secret in your UDF via the `dbutils.secrets.get` method.

  Example in Python:

  ```
  secretKey = dbutils.secrets.get(scope="myapp_secrets", key="secretKey")
  ```

  In a Java UDF, you could pass the value of the secret from Python as an argument or configure it in `spark.conf` like this:

  ```
  python
  spark.conf.set("myapp.secretKey", dbutils.secrets.get("myapp_secrets",
  "secretKey"))
  ```

3. Using Global Variables in Notebooks (if controlled by admin):

You could define certain global variables or environment variables directly within the notebooks by setting them at the beginning of the notebook, and the users simply use them without having permission to modify them.

Example:

```
python
# Define global constants
SECRET_KEY = "super_secret_value"
ENDPOINT_URL = "https://api.example.com"

# Use them in the notebook without users being able to modify them
print(f"Using secret key {SECRET_KEY}")
```

This approach, however, is less secure than using `spark.conf` or Databricks secrets.

4. Control Cluster Settings:

If you're managing a cluster that multiple users are accessing, you can set environment variables or configuration settings at the cluster level via the Databricks **cluster configuration** UI. For instance, under the "Advanced Options" > "Spark Config" section, you can add key-value pairs:

```
spark.myapp.secretKey=super_secret_value
```

These configuration settings can be accessed by UDFs and are not exposed to users.

Summary

- **`spark.conf.set`**: You can use Spark session configuration to set variables, which can be accessed within UDFs.
- **Databricks Secrets**: For sensitive information, you can securely store secrets using Databricks Secrets and retrieve them without exposing the values.

- **Cluster-level settings**: Set environment variables or configuration values at the cluster level for greater control over what users can access.

By using these methods, you ensure that the variables are set automatically for the users and cannot be modified directly.