Voltage Multi LOB Capabilities in UDP 3.1

For other information, please refer below pages:

- How Multi LOB works internally
- Decrypt data using AHM key and re-encrypting using Aetna key

Introduction

In order to comply with the Unified Data Platform (UDP) Data Protection standards, the use of encryption has been in place within the 2.6 UDP. Many of our existing tenants within the 2.6 UDP are already incorporating the use of Voltage User-Defined Functions (UDFs) within their ongoing ingestion, post-ingestion, and data use (egress) processes to decrypt the protected data when necessary and approved.

Unlike the 2.6 UDP where only Aetna Health Care Benefit data is present, the new 3.1 UDP environment now represents multiple lines of business. Aetna is now a CVS Company, and as such we now have the following lines of business within the 3.1 UDP:

- Health Care Benefits - **HCB** (Aetna)
- Retail - **RET** (CVS retail with front store and pharmacy)
- Pharmacy Benefit Management - **PBM**
- Enterprise - **ENT**

Some of the changes within the new 3.1 UDP, include:

- A simplified implementation and use of the UDFs
- Allowing more granular control of access to UDFs; which is now partially controlled by our tenants directly
- Capability to separate the access to data and UDFs between the lines of business
- Capability to mask certain data types where encryption alone may not provide adequate security measures
- A more than 200% performance improvement when invoking the UDFs

All of these changes are referred to as the ***Multi Line of Business*** capability

What Methods Are Used to Protect Data?

In the 3.1 UDP environments, there are 2 primary methods of data protection that are used:

- Encryption
- Masking

These methods employ different approaches to protecting data, and it's important to understand that the impact of these methods is different as well.

Encryption

Similar to the 2.6 UDP environment, the 3.1 UDP environments use format-preserving encryption methods (with the exception of the NOTES alias, please see below for more detail). This allows reliable encryption of various data types, and allows the ability to work with the encrypted values without the need for decryption capabilities. For example, 2 tables that contain the same data element - both encrypted with the same method - can still be joined within a query statement since the matching data values will also have matching encrypted values.

What has changed for the 3.1 UDP environment are the UDFs and the encryption keys that each UDF may use. This is important to understand as encrypted values from 2.6 UDP and 3.1 UDP cannot be compared. See below for more detail regarding the UDFs and keys.

An important aspect about encryption to understand is that the data is encrypted at rest. If a person or process were to select encrypted data from a source database and write it into a destination database, the data would remain encrypted and the underlying value of the data is not lost. If decryption privileges have been granted in the destination database, the value of the data can be revealed using the decryption UDFs. This is not the case with masking.

Masking

In some cases, encryption of a certain data type will either yield confusing results, or results that allow easy interpretation of the underlying data that we're intending to protect. In these situations, the data is either masked or encrypted and then masked. Common examples of data types where encryption alone is not a sufficient method of protecting the data include:

- Dates. Encrypted dates will still appear as valid dates formats. It is often not easy to know that the encrypted date is not the actual date value.
- Choice fields. Fields that only allow a very limited set of choices are not well-secured with encryption. Though the values will be encrypted, the encrypted results are repetitive and it can be relatively simple to determine the underlying value.

A key difference with masking compared to encryption is that the underlying value at rest is not masked - the masking of the value is applied during selection of the data. If a person or process were to select masked data from a source database and write it into a table in the destination database, the underlying data would be lost in the destination database as only the masked characters would be written into the destination database.

It is therefore absolutely critical to understand if data within the source database is masked when designing a process to move this data into a target database. Unmasking privileges must be obtained or the underlying data will not be moved into the destination.

How to Gain Decryption and/or Unmasking Privileges

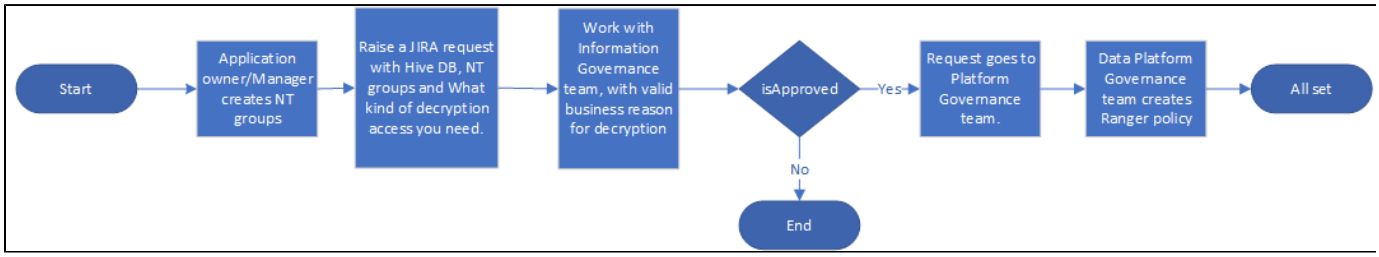
Unlike the 2.6 UDP environment, the UDFs in the 3.1 environment do not have to be loaded before each use, nor does a password file have to be maintained. The new UDFs are installed into the necessary database(s) for your use case, and made available through a Ranger policy.

The Ranger policies for UDF access make use of network groups that are created, maintained, and owned by your tenant team/department. Once the policies are available, your team will decide which processes and people may have the privileged access to use the UDFs. Since this is true for access to both encryption and decryption UDFs, we do require that separate groups be used for encryption and decryption so that only the minimum necessary processes and people have access to decryption capabilities.

Unmasking privileges are granted through a different set of policies that apply to an entire line of business, rather than a specific database. The policies for unmasking privileges (or more accurately, for preventing the masking of the data) make use of centrally-owned network groups. Once the necessary ID(s) have been added to the proper network group, the unmasking privilege will apply to any database within the specific line of business that the person or process may access.

To request creation of new UDF policies and/or unmasking capabilities, please submit a new Decryption Authorization Request. This process is designed to gather all of the details of your business use case, data needs, and the IDs and network groups that you'll use within the UDF policies.

Following is a high-level overview of the Decryption Authorization Request engagement:



Decryption Network Group Standards

Each tenant team/department is the owner of the decryption network groups used by the team. Each decryption UDF represents a different classification of data protection. For this reason, a different network group is required for each of the UDFs that your team wishes to access.

The network groups that are used for ingestion processes should not be the same as those used by post-ingestion and by data use processes (egress). The privilege to decrypt data is tightly-controlled and heavily audited, and we must only grant decryption access that complies with the overarching guidance of 'minimum necessary' when working with PHI/PII. Please see the related guidance for creating a network group to find the naming standards for any non-decryption network groups.

Following is the guidance for naming a network group that will be used to gain decryption UDF privileges:

1 letter prefix for the Line of Business: **P** for Pharmacy Benefit, **R** for retail, **H** for Aetna/Healthcare Benefit

+ 1 letter for the UDF classification: **C** for Confidential, **S** for SSNHICN, **N** for notes

+ 5 letters for the tenant team/department

+ 1-letter suffix for the UDP environment: **P** for Production, **Q** for QA, **D** for Development, **U** for UAT, **N** for other non-prod

Examples for PBM LOB Confidential:

1. In PBM line of business (P), Ingestion team's group (INGST) for Confidential decryption UDF access (C), within production environment (P) --> **PCINGSTP**
2. In PBM line of business (P), Ingestion team's group (INGST) for SSNHICN decryption UDF access (S), within QA environment (Q) --> **PN INGSTQ**

Other Network Group Considerations

It is required that each tenant team/department will have its own decryption network groups.

Network groups do not have to be specific to each database where decryption privileges have been requested or granted.

A team need only create the network group(s) needed for the specific decryption UDF(s) required. If use of only one of the UDFs will ever be required, there is no need to make the groups for all of the UDFs.

A team may not use its own network groups for unmasking privileges. The groups used within the unmasking policies are centrally-owned and managed by the Big Data & Development Organization (BD&D)

The New UDFs and Keys

While providing a single environment for all lines of business, we must ensure that access to the data and to the UDFs is only granted to a single line of business, with only approved exceptions allowing access across the lines of business (or to the Enterprise line of business). For this reason, changes were needed in order to properly manage the access and use of the UDFs within the 3.1 UDP.

UDF Classification	Healthcare Business HCB	Pharmacy benefit Management PBM	CVS Retail RET	Enterprise ENT

Confidential	encryptConfidentialhcb(param_1, 'ALIAS_HCB') decryptConfidentialhcb(param_1, 'ALIAS_HCB')	encryptConfidentialpbm(param_1, 'ALIAS_PBM') decryptConfidentialpbm(param_1, 'ALIAS_PBM')	encryptconfidentialret(param_1, 'ALIAS_RET') decryptconfidentialret(param_1, 'ALIAS_RET')	encryptconfidentialent(param_1, 'ALIAS_ENT') decryptconfidentialent(param_1, 'ALIAS_ENT')
SSNHICN	encryptssnhicnhcb(param_1, 'ALIAS_HCB') decryptssnhicnhcb(param_1, 'ALIAS_HCB')	encryptssnhicnpbm(param_1, 'ALIAS_PBM') decryptssnhicnpbm(param_1, 'ALIAS_PBM')	encryptssnhicnret(param_1, 'ALIAS_RET') decryptssnhicnret(param_1, 'ALIAS_RET')	encryptssnhicnent(param_1, 'ALIAS_ENT') decryptssnhicnent(param_1, 'ALIAS_ENT')
Notes	encryptnoteshcb(param_1, 'ALIAS_HCB') decryptnoteshcb(param_1, 'ALIAS_HCB')	encryptnotespbm(param_1, 'ALIAS_PBM') decryptnotespbm(param_1, 'ALIAS_PBM')	encryptnotesret(param_1, 'ALIAS_RET') decryptnotesret(param_1, 'ALIAS_RET')	encryptnotesent(param_1, 'ALIAS_ENT') decryptnotesent(param_1, 'ALIAS_ENT')

Aliases Used with Each UDF

Below is the list of UDF signatures (aliases) available in the Multi LOB version of encryption/decryption UDFs. The UDF signature and Alias names are suffixed with LOB values

Classification UDFs	Allowed Alias	Alias for Aetna(HCB)	Alias for Retail	Alias for PBM	Alias for
encryptConfidentialXXX decryptconfidentialXXX	1. NAME_XXX 2. ADDRESS_XXX 3. PHONEFAX_XXX 4. EMAIL_XXX 5. ACCOUNT_XXX 6. TIN_XXX 7. US7ASCII2_XXX 8. ALPHANUMERIC_XXX (only for decryption) 9. GEOLOC_XXX	1. NAME_HCB 2. ADDRESS_HCB 3. PHONEFAX_HCB 4. EMAIL_HCB 5. ACCOUNT_HCB 6. TIN_HCB 7. US7ASCII2_HCB 8. ALPHANUMERIC_HCB 9. GEOLOC_HCB	1. NAME_RET 2. ADDRESS_RET 3. PHONEFAX_RET 4. EMAIL_RET 5. ACCOUNT_RET 6. TIN_RET 7. US7ASCII2_RET 8. ALPHANUMERIC_RET 9. GEOLOC_RET	1. NAME_PBM 2. ADDRESS_PBM 3. PHONEFAX_PBM 4. EMAIL_PBM 5. ACCOUNT_PBM 6. TIN_PBM 7. US7ASCII2_PBM 8. ALPHANUMERIC_PBM 9. GEOLOC_PBM	1. NAM 2. ADD 3. PHC 4. EMA 5. ACC 6. TIN_ 7. US7 8. ALP 9. GEC
encryptssnhicnXXX decryptssnhicnXXX	1. SSNHICN_XXX 2. COMPOSITE_XXX	1. SSNHICN_HCB 2. COMPOSITE_HCB	1. SSNHICN_RET 2. COMPOSITE_RET	1. SSNHICN_PBM 2. COMPOSITE_PBM	1. SSN 2. COM
encryptnotesXXX decryptnotesXXX	1. NOTES_XXX	1. NOTES_HCB	1. NOTES_RET	1. NOTES_PBM	1. NOT

Alias Mapping to Data Categories

Please note: We make every attempt to keep this section current with regard to the UDP Data protection Standards. Please contact the BD&D Data Governance team in the event that a discrepancy is found between the standards and the following entries:

Sr. No	Data Categories	Alias for PBM	Alias for Retail	Alias for Aetna	Alias for ENT	Comments
1	Name, alias, preferred name, maiden	NAME_PBM	NAME_RET	NAME_HCB	NAME_ENT	
2	Street address (excludes city, state, zip)	ADDRESS_PBM	ADDRESS_RET	ADDRESS_HCB	ADDRESS_ENT	
3	Telephone numbers / fax numbers	PHONEFAX_PBM	PHONEFAX_RET	PHONEFAX_HCB	PHONEFAX_ENT	
4	Email address, Internet Protocol Address (customers), instant messenger name or user name	EMAIL_PBM	EMAIL_RET	EMAIL_HCB	EMAIL_ENT	
5	Social Security numbers & Medicare HICN	SSNHICN_PBM	SSNHICN_RET	SSNHICN_HCB	SSNHICN_ENT	

6	Account numbers, e.g. medical record numbers, bank account numbers, routing numbers	ACCOUNT_PBM US7ASCII2_PBM	ACCOUNT_RET US7ASCII2_RET	ACCOUNT_HCB US7ASCII2_HCB	ACCOUNT_ENT US7ASCII2_ENT	ACCOUNT for numeric-only formats US7ASCII2 for Alphanumeric and/or special characters
7	Race	Masking	Masking	Masking	Masking	
8	Ethnicity	Masking	Masking	Masking	Masking	
9	Language preference	Masking	Masking	Masking	Masking	
10	Health plan beneficiary numbers, policy number	ACCOUNT_PBM US7ASCII2_PBM	ACCOUNT_RET US7ASCII2_RET	ACCOUNT_HCB US7ASCII2_HCB	ACCOUNT_ENT US7ASCII2_ENT	ACCOUNT for numeric-only formats US7ASCII2 for Alphanumeric and/or special characters
11	Full face photographic image and any comparable image, finger print	BIO_PBM*	BIO_RET*	BIO_HCB*	BIO_ENT*	NOT YET IMPLEMENTED
12	License plate numbers	US7ASCII2_PBM	US7ASCII2_RET	US7ASCII2_HCB	US7ASCII2_ENT	
13	Driver's license or other government-issued ID number	US7ASCII2_PBM	US7ASCII2_RET	US7ASCII2_HCB	US7ASCII2_ENT	
14	Notes/Comments fields	NOTES_PBM *	NOTES_RET *	NOTES_HCB *	NOTES_ENT *	Use NOTES when you have XML, JSON and any freeform notes longer than 25 characters.
15	CVS/Aetna employee number	ACCOUNT_PBM	ACCOUNT_RET	ACCOUNT_HCB	ACCOUNT_ENT	ACCOUNT for numeric-only formats US7ASCII2 for Alphanumeric and/or special characters
16	Contact name	NAME_PBM	NAME_RET	NAME_HCB	NAME_ENT	
17	Tax ID / EIN	TIN_PBM	TIN_RET	TIN_HCB	TIN_ENT	
18	Date of Birth	Masking	Masking	Masking	Masking	
19	Geo Location	GEOLOC_PBM	GEOLOC_RET	GEOLOC_HCB	GEOLOC_ENT	
20	Medicare beneficiary Identifier (MBI)	US7ASCII2_PBM	US7ASCII2_RET	US7ASCII2_HCB	US7ASCII2_ENT	
21	Composite Key containing Confidential PII	US7ASCII2_PBM	US7ASCII2_RET	US7ASCII2_HCB	US7ASCII2_ENT	Fields generated from more than 1 piece of data, delimited or otherwise (e.g. EnrollmentDate-LastName-DOB)
22	Composite Key containing Restricted PII	COMPOSITE_PBM	COMPOSITE_RET	COMPOSITE_HCB	COMPOSITE_ENT	Fields generated from more than 1 piece of data, delimited or otherwise (e.g. EnrollmentDate-SSN-DOB)

* NOTES does not use Format-Preserving Encryption

What Is the Impact to the Data?

Comparing the 2.6 UDP Data Protection Requirements with the current requirements enabled in the 3.1 UDP, some subtle differences can be found.

Category	Data Element	Comment
Plan Sponsor	Plan Sponsor Name	No longer encrypted in 3.1
Plan Sponsor	Street Address	No longer encrypted in 3.1
Plan Sponsor	Phone/Fax	No longer encrypted in 3.1
Plan Sponsor	Email	No longer encrypted in 3.1
Member	Policy Number	New encryption for 3.1
Member	Health Plan Beneficiary Number	New encryption for 3.1

Member	Drivers License or other government-issued ID	New encryption for 3.1
Member	License Plate Number	New encryption for 3.1
Member	Notes or comments fields where pii elements may be stored	New encryption for 3.1
Member	Date of Birth	New masking for 3.1
Member	Language preference	New masking for 3.1
Member	Race	New masking for 3.1
Member	Ethnicity	New masking for 3.1
Employee	Employee ID or Employee Number	New encryption for 3.1

Production and Non-Production Encryption Keys

The non-production 3.1 UDP environments (HDPE_TEST, WIN_HATHI clusters) use Voltage QA servers within the AETHQ domain to maintain the encryption keys. The production 3.1 environment (MID_HATHI cluster) environment uses the Voltage production server within the AETH domain to maintain the encryption keys.

This means that the same data encrypted in different UDP 3.1 environments will produce different results and cannot be compared in their encrypted formats.

Length and Character Limitations

Please refer the below table, as this information for the 3.1 UDP is different than that for the 2.6 UDP.

Sr.No	Alias Name	internal encryption format	Minimum length in HDP 3.1	Max length in HDP 3.1	Input Character Set	Other Details
1	ACCOUNT	SSNHICNTIN-UpperOnly_str_eFPE-2	5	NA	42,48-57,65-90	Numeric values only
2	ADDRESS	NameAddrEmail_Upperonly_str_eFPE-2	9	NA	32,33,35,36,38-42,44-59,64-90,95,96	Supports alphanumeric and special characters
3	EMAIL	NameAddrEmail_Upperonly_str_eFPE-2	9	NA	32,33,35,36,38-42,44-59,64-90,95,96	Supports alphanumeric and special characters
4	NAME	NameAddrEmail_Upperonly_str_eFPE-2	9	NA	32,33,35,36,38-42,44-59,64-90,95,96	Supports alphanumeric and special characters
5	PHONEFAX	NameAddrEmail_Upperonly_str_eFPE-2	5	NA	32,33,35,36,38-42,44-59,64-90,95,96	
6	SSNHICN	SSNHICNTIN-UpperOnly_str_eFPE-2	5	19	42,48-57,65-90	
7	TIN	SSNHICNTIN-UpperOnly_str_eFPE-2	5	NA	42,48-57,65-90	
8	US7ASCII2	US7ASCII-PRINTABLE-2	6	50	32-126	Supports alphanumeric and special characters
9	COMPOSITE	US7ASCII-PRINTABLE-2	6	50	32-126	Supports alphanumeric and special characters

Data Type Limitations

The UDFs will encrypt and decrypt if the data/column type is String. If the source column is other than String, the data must be type cast as indicated below.

```
select lastname, encryptconfidentialhcb(CAST(lastname as string), 'NAME_HCB') from
employee;
```

Query With/Without use clause

1) With "**use**" clause

```
use dbName;

select lastname, encryptconfidentialhcb(lastname, 'NAME_HCB') from employee;
```

Note: Assuming both UDF and table exist in the same database

(OR)

2) Without "**use**" clause

```
select lastname, dbName.encryptconfidentialhcb(lastname, 'NAME_HCB') from dbName.employee;
```

Query Execution Examples

Simple

LOB/ Query Type	Select and encrypt

HCB	`select lastname, encryptconfidentialhcb(lastname, 'NAME_HCB'), address, encryptconfidentialhcb(address, 'ADDRESS_HCB'), email, encryptconfidentialhcb(email, 'EMAIL_HCB'), phone, encryptconfidentialhcb(phone, 'PHONEFAX_HCB'), tin, encryptconfidentialhcb(tin, 'TIN_HCB') from voltage_udf_test limit 5;`
	
	`select ssn, encryptssnhicnhcb(ssn, 'SSNHICN_HCB') from voltage_udf_test limit 5;`
	
	`select notes, encryptnoteshcb(notes, 'NOTES_HCB') from voltage_udf_test limit 5;`

PBM	`select lastname, encryptconfidentialpbm(lastname, 'NAME_PBM'), address, encryptconfidentialpbm(address, 'ADDRESS_PBM'), email, encryptconfidentialpbm(email, 'EMAIL_PBM'), phone, encryptconfidentialpbm(phone, 'PHONEFAX_PBM'), tin, encryptconfidentialpbm(tin, 'TIN_PBM') from voltage_udf_test limit 5;`
	
	`select ssn, encryptssnhicnpbm(ssn, 'SSNHICN_PBM') from voltage_udf_test limit 5;`
	`select notes, encryptnotespbm(notes, 'NOTES_PBM') from voltage_udf_test limit 5;`

RET	`select lastname, encryptconfidentialret(lastname, 'NAME_RET'), address, encryptconfidentialret(address, 'ADDRESS_RET'), email, encryptconfidentialret(email, 'EMAIL_RET'), phone, encryptconfidentialret(phone, 'PHONEFAX_RET'), tin, encryptconfidentialret(tin, 'TIN_RET') from voltage_udf_test limit 5;`
	
	`select ssn, encryptssnhicnret(ssn, 'SSNHICN_RET') from voltage_udf_test limit 5;`
	`select notes, encryptnotesret(notes, 'NOTES_RET') from voltage_udf_test limit 5;`

ENT	`select lastname, encryptconfidentialent(lastname, 'NAME_ENT'), address, encryptconfidentialent(address, 'ADDRESS_ENT'), email, encryptconfidentialent(email, 'EMAIL_ENT'), phone, encryptconfidentialent(phone, 'PHONEFAX_ENT'), tin, encryptconfidentialent(tin, 'TIN_ENT') from gladiators.voltage_udf_test limit 5;`  `select ssn, encryptssnhicnent(ssn, 'SSNHICN_ENT') from voltage_udf_test limit 5;` `select notes, encryptnotesent(notes, 'NOTES_ENT') from voltage_udf_test limit 5;`

More Complex

Complex Type	Query

CREATE New Table from existing table	**CREATE new table** `CREATE table complex_create_query_test as SELECT lastname , encryptconfidentialhcb(lastname, 'NAME_HCB'), address, encryptconfidentialhcb(address, 'ADDRESS_HCB'), email , encryptconfidentialhcb(email, 'EMAIL_HCB'), phone , encryptconfidentialhcb(phone, 'PHONEFAX_HCB') FROM voltage_udf_test;`
INSERT Query	**INSERT Query** `INSERT INTO complex_query_test SELECT lastname , encryptconfidentialhcb(lastname, 'NAME_HCB'), address, encryptconfidentialhcb(address, 'ADDRESS_HCB'), email , encryptconfidentialhcb(email, 'EMAIL_HCB'), phone , encryptconfidentialhcb(phone, 'PHONEFAX_HCB') FROM voltage_udf_test;`
PARTITION Query	**PARTITION while create** `CREATE table complex_partition_query_test (lastname String, encryptedname String) PARTITIONED BY (p_date String); INSERT INTO TABLE complex_partition_query_test PARTITION(p_date) SELECT name, encryptconfidentialhcb(lastname,'NAME_HCB'), '20191011' FROM voltage_udf_test;`
JOIN Query	**JOIN** `CREATE table complex_join_query_test as SELECT a.lastname, encryptconfidentialhcb(b.lastname, 'NAME_HCB') FROM voltage_udf_test a JOIN voltage_udf_test b ON (a.lastname = b.lastname);`

Using Spark

We can use the Hive Warehouse Connector (HWC) API to access any type of table and the UDF's in the Hive catalog from Spark.

Using PySpark

```
pyspark --jars
/usr/hdp/3.1.4.0-315/hive_warehouse_connector/hive-warehouse-connector-assembly-1.0.0.
3.1.4.0-315.jar
 --py-files
/usr/hdp/3.1.4.0-315/hive_warehouse_connector/pyspark_hwc-1.0.0.3.1.4.0-315.zip
 --queue encrypt
 --conf
spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://xhadzkw1p.aetna.com:2181,xhadzkw2p.a
etna.com:2181,xhadzkw3p.aetna.com:2181,xhadzkw4p.aetna.com:2181,xhadzkw5p.aetna.com:21
81/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"

>>> from pyspark_llap import HiveWarehouseSession
>>> hive = HiveWarehouseSession.session(spark).build()
>>> hive.showDatabases().show()
+-----------------+
|    database_name|
+-----------------+
|          default|
|       gladiators|
|      phoenix_dev|
|poc_benchmark_dev|
|   ranger_support|
+-----------------+
>>> hive.setDatabase("gladiators")
>>> d=hive.execute("SELECT lastname, encryptconfidentialpbm(lastname, 'NAME_PBM') as
encryptedName FROM voltage_udf_test limit 5")
>>> d.show()
+--------+-------------+
|lastname|encryptedname|
+--------+-------------+
|   White|    3JB1FW@#s|
|  Borden|    grtzEkKmg|
|   Green|    O__?~Ww/s|
|  Munsch|    FvE2_mC1c|
|  Aragon|    _oAm64Szw|
+--------+-------------+
```

Using Scala:

```
spark-shell --jars
/usr/hdp/current/hive_warehouse_connector/hive-warehouse-connector-assembly-1.0.0.3.1.
4.0-315.jar
 --conf
spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://xhadzkw1p.aetna.com:2181/;serviceDis
coveryMode=zooKeeper;zooKeeperNamespace=hiveserver2"
 --queue encrypt

scala> import com.hortonworks.hwc.HiveWarehouseSession
scala> import com.hortonworks.hwc.HiveWarehouseSession._
scala> val hive = HiveWarehouseSession.session(spark).build()
scala> hive.showDatabases().show()
+-----------------+
|    database_name|
+-----------------+
|          default|
|       gladiators|
|      phoenix_dev|
|poc_benchmark_dev|
|   ranger_support|
+-----------------+
scala> hive.setDatabase("gladiators")
scala> hive.execute("SELECT * FROM voltage_udf_test limit 5").show()
scala> val d=hive.execute("SELECT lastname, encryptconfidentialpbm(lastname,
'NAME_PBM') as encryptedName FROM voltage_udf_test limit 5")
scala> d.show()
+--------+-------------+
|lastname|encryptedname|
+--------+-------------+
|   White|    3JB1FW@#s|
|  Borden|    grtzEkKmg|
|   Green|    O__?~Ww/s|
|  Munsch|    FvE2_mC1c|
|  Aragon|    _oAm64Szw|
+--------+-------------+
```

Using Spark in Java

Github Code: https://github.aetna.com/dpe-fw-common/hdp3-1-sandbox/tree/develop/spark-hive-job

Execute Java Spark from client mode:

```
spark-submit --class com.aetna.dpe.SparkHiveWarehouseConnectionApp \
 --master yarn \
 --deploy-mode client \
 --jars hive-warehouse-connector_2.11-1.0.0.3.0.1.0-187.jar \
 --conf spark.security.credentials.hiveserver2.enabled=false \
 --conf
spark.sql.hive.hiveserver2.jdbc.url="jdbc:hive2://xhadzkw1p.aetna.com:2181,xhadzkw2p.a
etna.com:2181,xhadzkw3p.aetna.com:2181,xhadzkw4p.aetna.com:2181,xhadzkw5p.aetna.com:21
81/;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2" \
 --driver-memory 4g \
 --executor-memory 2g \
 --executor-cores 1 \
spark-hive-job-0.0.1-SNAPSHOT.jar \
 10
```

Using Native Spark with RDD

At this moment there is not support for this. We are working on it, but there is no ETA yet.