Protegrity Big Data Protector Guide 9.2.0.0

Created on: Aug 8, 2024

Notice

Copyright

Copyright © 2004-2024 Protegrity Corporation. All rights reserved.

Protegrity products are protected by and subject to patent protections;

Patent: *https://www.protegrity.com/patents*.

Protegrity logo is the trademark of Protegrity Corporation.

NOTICE TO ALL PERSONS RECEIVING THIS DOCUMENT

Some of the product names mentioned herein are used for identification purposes only and may be trademarks and/or registered trademarks of their respective owners.

Windows, Azure, MS-SQL Server, Internet Explorer and Internet Explorer logo, Active Directory, and Hyper-V are registered trademarks of Microsoft Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SCO and SCO UnixWare are registered trademarks of The SCO Group.

Sun, Oracle, Java, and Solaris are the registered trademarks of Oracle Corporation and/or its affiliates in the United States and other countries.

Teradata and the Teradata logo are the trademarks or registered trademarks of Teradata Corporation or its affiliates in the United States and other countries.

Hadoop or Apache Hadoop, Hadoop elephant logo, Hive, Presto, and Pig are trademarks of Apache Software Foundation.

Cloudera and the Cloudera logo are trademarks of Cloudera and its suppliers or licensors.

Hortonworks and the Hortonworks logo are the trademarks of Hortonworks, Inc. in the United States and other countries.

Greenplum Database is the registered trademark of VMware Corporation in the U.S. and other countries.

Pivotal HD is the registered trademark of Pivotal, Inc. in the U.S. and other countries.

PostgreSQL or Postgres is the copyright of The PostgreSQL Global Development Group and The Regents of the University of California.

AIX, DB2, IBM and the IBM logo, and z/OS are registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Utimaco Safeware AG is a member of the Sophos Group.

Xen, XenServer, and Xen Source are trademarks or registered trademarks of Citrix Systems, Inc. and/or one or more of its subsidiaries, and may be registered in the United States Patent and Trademark Office and in other countries.

VMware, the VMware “boxes” logo and design, Virtual SMP and VMotion are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions.

Amazon Web Services (AWS) and AWS Marks are the registered trademarks of Amazon.com, Inc. in the United States and other countries.

HP is a registered trademark of the Hewlett-Packard Company.

HPE Ezmeral Data Fabric is the trademark or registered trademark of Hewlett Packard Enterprise in the United States and other countries.

Dell is a registered trademark of Dell Inc.

Novell is a registered trademark of Novell, Inc. in the United States and other countries.

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

Mozilla and Firefox are registered trademarks of Mozilla foundation.

Chrome and Google Cloud Platform (GCP) are registered trademarks of Google Inc.

Table of Contents

Copyright......2

Chapter 1 Introduction to This Guide......6
1.1 Sections contained in this Guide......6
1.2 Accessing the Protegrity documentation suite......6
1.2.1 Viewing product documentation......6
1.2.2 Downloading product documentation......7

Chapter 2 Overview of the Big Data Protector......9
2.1 Components of Hadoop......10
2.1.1 Hadoop Distributed File System (HDFS)......10
2.1.2 MapReduce......10
2.1.3 Hive......11
2.1.4 Pig......11
2.1.5 HBase......11
2.1.6 Impala......11
2.1.7 Spark......11
2.2 Features of the Protegrity Big Data Protector......11
2.3 Using Protegrity Data Security Platform with Hadoop......13
2.4 Overview of Hadoop Application Protection......14
2.4.1 Protection in MapReduce Jobs......14
2.4.2 Protection in Hive Queries......14
2.4.3 Protection in Pig Jobs......14
2.4.4 Protection in HBase......14
2.4.5 Protection in Impala......15
2.4.6 Protection in Spark......15
2.5 Data Security Policy and Protection Methods......15
2.6 Installing and Uninstalling Big Data Protector......16
2.7 Understanding the Architecture......16
2.8 Working with the Log Forwarder......16
2.8.1 Logging Architecture......17
2.8.2 Logging Architecture of the Big Data Protector Cluster without the Proxy......17
2.8.3 Logging Architecture of the Big Data Protector Cluster with the Proxy......18

Chapter 3 Hadoop Application Protector......19
3.1 Using the Hadoop Application Protector......19
3.2 Prerequisites......19
3.3 MapReduce APIs......20
3.4 Sample Code Usage......20
3.4.1 Main Job Class – ProtectData.java......20
3.4.2 Mapper Class – ProtectDataMapper.java......21
3.5 Hive UDFs......23
3.6 Pig UDFs......23

Chapter 4 HBase......24
4.1 Overview of the HBase Protector......24
4.2 HBase Protector Usage......24
4.3 Adding Data Elements and Column Qualifier Mappings to a New Table......25
4.4 Adding Data Elements and Column Qualifier Mappings to an Existing Table......25
4.5 Inserting Protected Data into a Protected Table......26
4.6 Retrieving Protected Data from a Table......26
4.7 HBase Commands......27

4.8 Ingesting Data Securely....................27
4.9 Extracting Data Securely....................27

Chapter 5 Impala....................28
5.1 Overview of the Impala Protector....................28
5.2 Impala Protector Usage....................28
5.2.1 Creating the */user/impala* path in Impala with Supergroup Permissions....................28
5.3 Impala UDFs....................29
5.4 Inserting Data from a File into a Table....................29
5.4.1 Preparing the environment for the *basic_sample.csv* file....................29
5.4.2 Populating the table *sample_table* from the *basic_sample_data.csv* file....................29
5.5 Protecting Existing Data....................30
5.6 Unprotecting the Protected Data....................31
5.7 Retrieving Data from a Table....................31

Chapter 6 Spark....................32
6.1 Overview of the Spark Protector....................32
6.2 Spark Protector Usage....................32
6.3 Spark Java....................33
6.3.1 Spark Java APIs....................33
6.3.2 Spark APIs and Supported Protection Methods....................33
6.3.3 Loading the Cleartext Data from a File to HDFS....................34
6.3.4 Protecting the Existing Data....................35
6.3.5 Unprotecting the Protected Data....................35
6.3.6 Retrieving the Unprotected Data from a File....................36
6.4 Spark SQL....................36
6.4.1 DataFrames....................36
6.4.2 SQLContext....................36
6.4.3 Spark SQL UDFs....................37
6.4.4 Inserting Data from a File into a Table....................37
6.4.5 Protecting Existing Data....................37
6.4.6 Unprotecting and Viewing the Protected Data....................38
6.4.7 Retrieving Data from a Table....................38
6.4.8 Calling Spark SQL UDFs from Domain Specific Language (DSL)....................39
6.5 Spark Scala....................41
6.5.1 Sample Code Usage for Spark (Scala)....................41
6.5.1.1 Main Job Class for Protect Operation – ProtectData.scala....................41
6.5.1.2 Main Job Class for Unprotect Operation – UnProtectData.scala....................42
6.5.1.3 Utility to call Protect or Unprotect Function – DataLoader.scala....................42
6.5.1.4 ProtectFunction.scala....................43
6.5.1.5 UnprotectFunction.scala....................44

Chapter 7 Appendix: Return Codes....................45

Appendix 8 Appendix: Migrating Tokenized Unicode Data from and to a Teradata Database....................49
8.1 Migrating Tokenized Unicode Data from a Teradata Database....................49
8.1.1 Migrating Tokenized Unicode data from Teradata database to Hive or Impala and unprotecting it using Hive or Impala protector....................49
8.1.2 Migrating Tokenized Unicode data from a Teradata database to Hadoop and Unprotecting it using MapReduce or Spark protector....................50
8.2 Migrating Tokenized Unicode Data to a Teradata Database....................50
8.2.1 Migrating Tokenized Unicode data using Hive or Impala protector to Teradata database....................51
8.2.2 Protecting Unicode data using MapReduce or Spark protector and Migrating it to a Teradata database....................51

Chapter 1

Introduction to This Guide

1.1 Sections contained in this Guide

1.2 Accessing the Protegrity documentation suite

This guide provides information about configuring and using the Protegrity Big Data Protector (BDP) for Hadoop.

This guide should be used along with the *Protegrity Enterprise Security Administrator Guide 9.2.0.0*, which explains the mechanism of managing the data security policy.

It is recommended that you first read the sections explaining the basics of Big Data Protector in this guide.

1.1 Sections contained in this Guide

This section provides a short description about the sections contained in this guide.

The guide is broadly divided into the following sections:

- Section *Introduction to This Guide* defines the purpose and scope for this guide. In addition, it explains how information is organized in this guide.
- Section *Overview of the Big Data Protector* provides a general idea of Hadoop and how it has been integrated with the Big Data Protector. In addition, it describes the protection coverage of various Hadoop ecosystem applications, such as MapReduce, Hive and Pig.
- Section *Hadoop Application Protector* provides information about Hadoop Application Protector.
- Section *HBase* provides information about the Protegrity HBase protector.
- Section *Impala* provides information about the Protegrity Impala protector.
- Section *Spark* provides information about the Protegrity Spark Java and Spark SQL protectors. In addition, it provides information about Spark Scala.
- Section *Appendix: Return Codes* provides information about all possible error codes and error descriptions for Big Data Protector.
- Section *Appendix: Migrating Tokenized Unicode Data from and to a Teradata Database* describes procedures for migrating tokenized Unicode data from and to a Teradata database.

1.2 Accessing the Protegrity documentation suite

This section describes the methods to access the *Protegrity Documentation Suite* using the *My.Protegrity* portal.

1.2.1 Viewing product documentation

The **Product Documentation** section under **Resources** is a repository for Protegrity product documentation. The documentation for the latest product release is displayed first. The documentation is available in the HTML format and can be viewed using your browser. You can also view and download the *.pdf* files of the required product documentation.

1. Log in to the *My.Protegrity* portal.
2. Click **Resources** > **Product Documentation**.
3. Click a product version.
   The documentation appears.

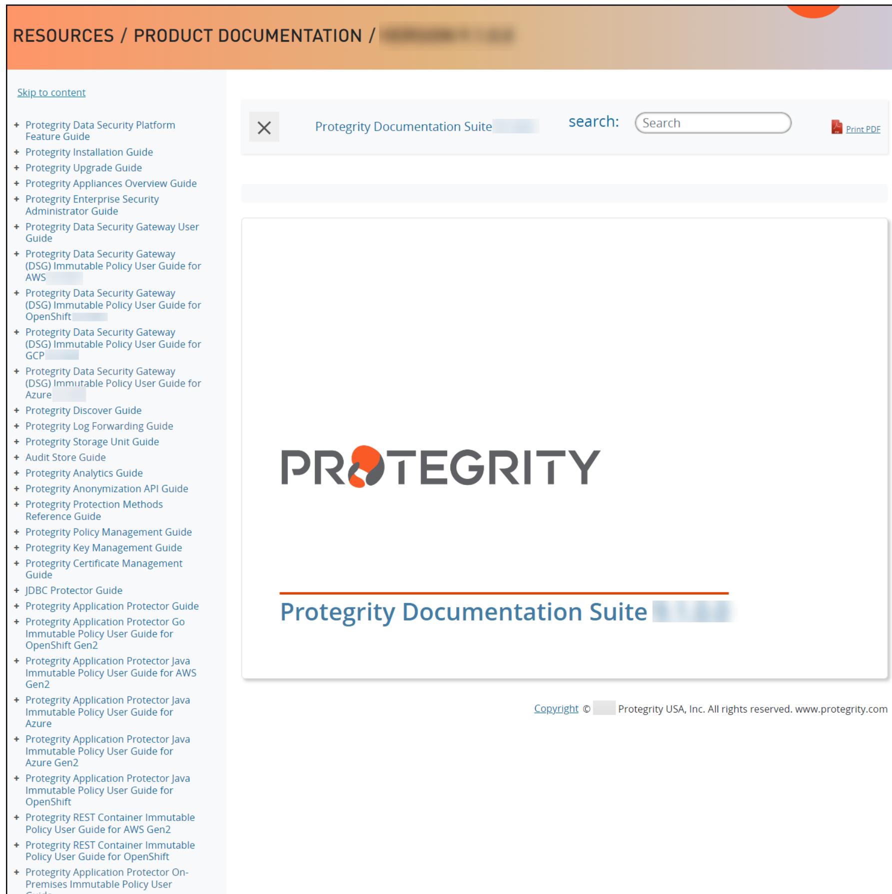

Figure 1-1: Documentation

4. Expand and click the link for the required documentation.
5. If required, then enter text in the **Search** field to search for keywords in the documentation.
   The search is dynamic, and filters results while you type the text.
6. Click the **Print PDF** icon from the upper-right corner of the page.
   The page with links for viewing and downloading the guides appears. You can view and print the guides that you require.

1.2.2 Downloading product documentation

This section explains the procedure to download the product documentation from the *My.Protegrity* portal.

1. Click **Product Management** > **Explore Products**.
2. Select **Product Documentation**.
   The **Explore Products** page is displayed. You can view the product documentation of various Protegrity products as per their releases, containing an overview and other guidelines to use these products at ease.
3. Click **View Products** to advance to the product listing screen.
4. Click the **View** icon ( ) from the **Action** column for the row marked **On-Prem** in the **Target Platform Details** column.
   If you want to filter the list, then use the filters for: **OS**, **Target Platform**, and **Search** fields.

5. Click the icon for the action that you want to perform.

Chapter 2

Overview of the Big Data Protector

2.1 Components of Hadoop
2.2 Features of the Protegrity Big Data Protector
2.3 Using Protegrity Data Security Platform with Hadoop
2.4 Overview of Hadoop Application Protection
2.5 Data Security Policy and Protection Methods
2.6 Installing and Uninstalling Big Data Protector
2.7 Understanding the Architecture
2.8 Working with the Log Forwarder

The Protegrity Big Data Protector was the first to support fine grained data protection on Hadoop and enterprise-ready Hadoop native encryption.

Protegrity has regularly updated Big Data Protector to include support for MapReduce, Hive, Pig, HBase, Impala, and Spark.

Starting from the Big Data Protector, version 7.0, which released in 2017, support for native installers and rolling restarts for the Cloudera and Ambari environments is added.

Protegrity also supports Spark SQL, Spark streaming, Kafka, and Flume type of real-time ingestion tools. You can achieve both, batch and real time data protection, using Protegrity Big Data Protector.

The following are some of the use cases that the Protegrity Big Data Protector satisfies:

- Data protection at source applications: In this case, the sensitive data is fully protected wherever it flows, including the Hadoop ecosystem.

  In addition, it ensures that the Hadoop system, which stores the protected data, is not brought into scope for PCI, PII, GDPR, HIPPA, FIPS, and other compliance policies.

In the Protegrity Big Data Protector for Apache Hadoop, the data is split and shared with all the data nodes in the Hadoop cluster. The Big Data Protector is deployed on each of these nodes where the protection enforcement policies are shared.

The Protegrity Big Data Protector is scalable and new nodes can be added as required. It is cost effective since massively parallel computing is done on commodity servers, and it is flexible as it can work with data from any number of sources. The Big Data Protector is fault tolerant as the system redirects the work to another node if a node is lost. It can handle structured data irrespective of their native formats.

The Big Data Protector protects data, which is handled by various Hadoop applications and protects files stored in the cluster. MapReduce, Hive, Pig, HBase, Spark, and Impala can use Protegrity protection interfaces to protect data as it is stored or retrieved from the Hadoop cluster. All standard protection techniques offered by Protegrity are applicable to Big Data Protector.

For more information about the available protection options, such as data types, Tokenization or Encryption types, or length preserving and non-preserving tokens, refer to *Protection Methods Reference Guide 9.2.0.0*.

2.1 Components of Hadoop

The Big Data Protector works on the Hadoop framework as shown in the following figure.

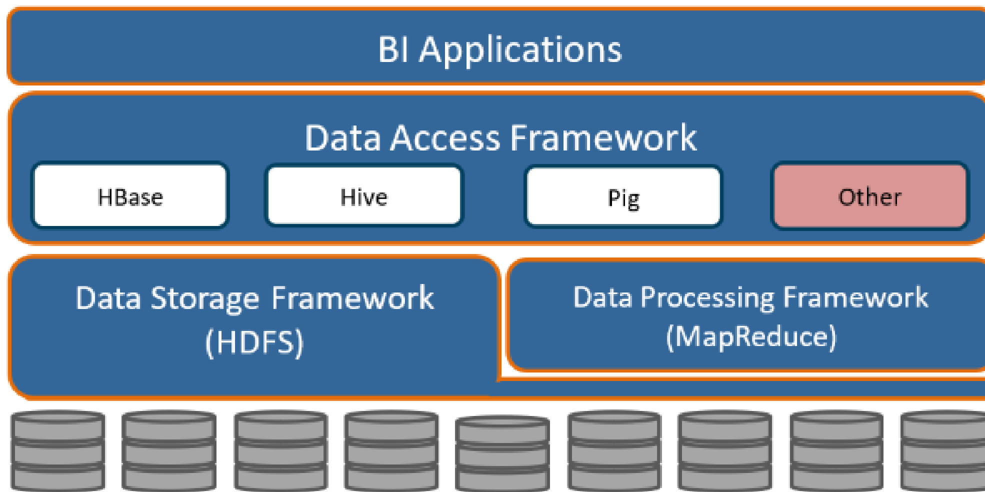

Figure 2-1: Hadoop Components

> **Note:**
> The illustration of Hadoop components is an example.
>
> Based on requirements, the components of Hadoop might be different.

Hadoop interfaces have been used extensively to develop the Big Data Protector. It is a common deployment practice to utilize Hadoop Distributed File System (HDFS) to store the data, and let MapReduce process the data and store the result back in HDFS.

2.1.1 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) spans across all nodes in a Hadoop cluster for data storage. It links together the file systems on many nodes to make them into one big file system. HDFS assumes that nodes will fail, so data is replicated across multiple nodes to achieve reliability.

2.1.2 MapReduce

The MapReduce framework assigns work to every node in large clusters of commodity machines. MapReduce programs are sets of instructions to parse the data, create a map or index, and aggregate the results. Since data is distributed across multiple nodes, MapReduce programs run in parallel, working on smaller sets of data.

A MapReduce job is executed by splitting each job into small Map tasks, and these tasks are executed on the node where a portion of the data is stored. If a node containing the required data is saturated and not able to execute a task, then MapReduce shifts the task to the least busy node by replicating the data to that node. A Reduce task combines results from multiple Map tasks, and store all of them back to the HDFS.

2.1.3 Hive

The Hive framework resides above Hadoop to enable ad hoc queries on the data in Hadoop. Hive supports HiveQL, which is similar to SQL. Hive translates a HiveQL query into a MapReduce program and then sends it to the Hadoop cluster.

2.1.4 Pig

Pig is a high-level platform for creating MapReduce programs used with Hadoop.

2.1.5 HBase

HBase is a column-oriented datastore, meaning it stores data by columns rather than by rows. This makes certain data access patterns much less expensive than with traditional row-oriented relational database systems. The data in HBase is protected transparently using Protegrity HBase coprocessors.

2.1.6 Impala

Impala is an MPP SQL query engine for querying the data stored in a cluster. It provides the flexibility of the SQL format and is capable of running the queries on HDFS in HBase.

The Impala daemon runs on each node in the cluster, reading and writing to data in the files, and accepts queries from the Impala shell command. The following are the core components of Impala:

- Impala daemon (*impalad*) – This component is the Impala daemon which runs on each node in the cluster. It reads and writes the data in the files and accepts queries from the Impala shell command.
- Impala Statestore (*statestored*) – This component checks the health of the Impala daemons on all the nodes contained in the cluster. If a node is unavailable due to any error or failure, then the Impala *statestore* component informs all other nodes about the failed node to ensure that new queries are not sent to the failed node.
- Impala Catalog (*catalogd*) – This component is responsible for communicating any changes in the metadata received from the Impala SQL statements to all the nodes in the cluster.

2.1.7 Spark

Spark is an execution engine that carries out batch processing of jobs in-memory and handles a wider range of computational workloads. In addition to processing a batch of stored data, Spark is capable of manipulating data in real time.

Spark leverages the physical memory of the Hadoop system and utilizes Resilient Distributed Datasets (RDDs) to store the data in-memory and lowers latency, if the data fits in the memory size. The data is saved on the hard drive only if required.

2.2 Features of the Protegrity Big Data Protector

The Protegrity Big Data Protector (Big Data Protector) uses patent-pending vaultless tokenization and central policy control for access management and secures sensitive data at rest in the following areas:

- Data in HDFS
- Data used during MapReduce, Hive and Pig processing, and with HBase, Impala, and Spark
- Data traversing enterprise data systems

The data is protected from internal and external threats, and users and business processes can continue to utilize the secured data.

Data protection may be by encryption or tokenization. In tokenization, data is converted to similar looking inert data known as tokens where the data format and type can be preserved. These tokens can be detokenized back to the original values when it is required.

Protegrity secures files with volume encryption and also protects data inside files using tokenization and strong encryption protection methods. Depending on the user access rights and the policies set using Policy management in ESA, this data is unprotected.

The Protegrity Hadoop Big Data Protector provides the following features:

- Provides fine grained field-level protection within the MapReduce, Hive, Pig, HBase, and Spark frameworks.
- Provides directory and file level protection (encryption).
- Provides Protegrity Format Preserving Encryption (FPE) method for structured data. The following data types are supported:
  - Numeric (0-9)
  - Alpha (a-z, A-Z)
  - Alpha-Numeric (0-9, a-z, A-Z)
  - Credit Card (0-9)
  - Unicode Basic Latin and Latin-1 Supplement Alpha
  - Unicode Basic Latin and Latin-1 Supplement Alpha-Numeric

  For more information about FPE, refer to *Protection Methods Reference Guide 9.2.0.0*.
- Retains distributed processing capability as field-level protection is applied to the data.
- Protects data in the Hadoop cluster using role-based administration with a centralized security policy.
- Starting from the Big Data Protector, version 7.0 release, native installers for the Cloudera and Ambari environments are being provided. These new installers simplify the task of installing, configuring, and managing Big Data Protector using Cloudera Manager or the Ambari UI.
- Simplified installation, administration, and management of Big Data Protector using the following components:
  - Parcels: In Cloudera Manager, a Big Data Protector Parcel, which is a single consolidated file, contains all the required files for installing and using Big Data Protector on a cluster and the metadata used by Cloudera Manager.
  - Custom Service Descriptors (CSDs): In Cloudera Manager, a CSD contains all the configurations required to describe and manage the Big Data Protector services. The CSDs are provided as Jar files.
  - Management Packs: In Ambari, a Big Data Protector management pack, which is a single consolidated file, contains all the required files for installing and using Big Data Protector on a cluster and the metadata used by the Ambari UI.
- Easy monitoring of the Big Data Protector services, such as, BDP PEP, using the Cloudera Manager UI instead of the CLI.
- Easy monitoring of the Big Data Protector services, such as BDP PEP and BDP HDFSFP, using Ambari instead of the CLI.
- Automatic deactivation of older Big Data Protector parcels in Cloudera Manager on update.
- Provides logging and viewing data access activities and real-time alerts with a centralized monitoring system.
- Ensures minimal overhead for processing secured data, with minimal consumption of resources, threads and processes, and network bandwidth.
- Provides transparent data protection with Protegrity HBase protectors.

> **Note:**
> The following figure illustrates the various components in an Enterprise Hadoop ecosystem.

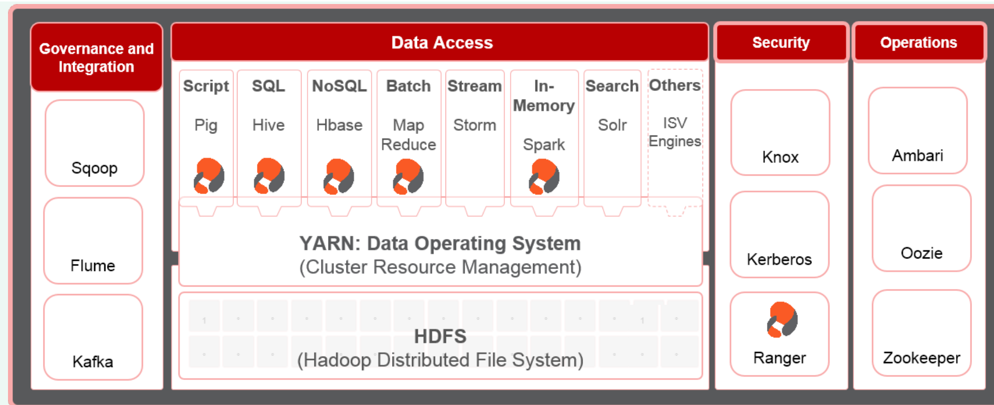

Figure 2-2: Enterprise Hadoop Components

Currently, Protegrity supports MapReduce, Hive, Pig, and HBase which utilize HDFS as the data storage layer. The following points can be referred to as general guidelines:

- Sqoop: Sqoop can be used for ingestion into HDFSFP protected zone (For Hortonworks, Cloudera, and Pivotal HD).
- Beeline and Hue on Cloudera: Beeline and Hue are certified with the Hive protector.
- Beeline and Hue on Hortonworks & Pivotal HD: Beeline and Hue are certified with the Hive protector.
- Ranger (Hortonworks): Ranger is certified to work with the Hive protector.
- Sentry (Cloudera): Sentry is certified with the Hive protector and Impala protector.
- Spark
- Impala

We neither support nor have certified other components in the Hadoop stack. We strongly recommend consulting Protegrity, before using any unsupported components from the Hadoop ecosystem with our products.

2.3 Using Protegrity Data Security Platform with Hadoop

To protect data, the components of the Protegrity Data Security Platform are integrated into the Hadoop cluster.

The Enterprise Security Administrator (ESA) is a soft appliance that needs to be pre-installed on a separate server, which is used to create and manage policies.

For more information about installing the ESA, and creating and managing policies, refer to *Installation Guide 9.2.0.0* and *Policy Management Guide 9.2.0.0* respectively.

Each task runs on a node under the same Hadoop user. Every user has a policy deployed for running their jobs on this system. Hadoop manages the accounts and users. You can get the Hadoop user information from the actual job configuration.

HDFS implements a permission model for files and directories, based on the Portable Operating System Interface (POSIX) for Unix model. Each file and directory is associated with an owner and a group. Depending on the permissions granted, users for the file and directory can be classified into one of these three groups:

- Owner
- Other users of the group
- All other users

2.4 Overview of Hadoop Application Protection

This section describes the various levels of protection provided by Hadoop Application Protection.

2.4.1 Protection in MapReduce Jobs

A MapReduce job in the Hadoop cluster involves sensitive data. You can use Protegrity interfaces to protect data when it is saved or retrieved from a protected source. The output data written by the job can be encrypted or tokenized. The protected data can be subsequently used by other jobs in the cluster in a secured manner. Field level data can be secured and ingested into HDFS by independent Hadoop jobs or other ETL tools.

For more information about secure ingestion of data in Hadoop, refer to section *Ingesting Files Using Hive Staging*.

For more information on the list of available APIs, refer to section *MapReduce APIs*.

If Hive queries are created to operate on sensitive data, then you can use Protegrity Hive UDFs for securing data. While inserting data to Hive tables, or retrieving data from protected Hive table columns, you can call Protegrity UDFs loaded into Hive during installation. The UDFs protect data based on the input parameters provided.

Secure ingestion of data into HDFS to operate Hive queries can be achieved by independent Hadoop jobs or other ETL tools.

For more information about securely ingesting data in Hadoop, refer to section *Ingesting Data Securely*.

2.4.2 Protection in Hive Queries

Protection in Hive queries is done by Protegrity Hive UDFs, which translates a HiveQL query into a MapReduce program and then sends it to the Hadoop cluster.

For more information on the list of available UDFs, refer to section *Hive UDFs*.

2.4.3 Protection in Pig Jobs

Protection in Pig jobs is done by Protegrity Pig UDFs, which are similar in function to the Protegrity UDFs in Hive.

For more information on the list of available UDFs, refer to section *Pig UDFs*.

2.4.4 Protection in HBase

HBase is a database which provides random read and write access to tables, consisting of rows and columns, in real-time. HBase is designed to run on commodity servers, to automatically scale as more servers are added, and is fault tolerant as data is divided across servers in the cluster. HBase tables are partitioned into multiple regions. Each region stores a range of rows in the table. Regions contain a datastore in memory and a persistent datastore(HFile). The Name node assigns multiple regions to a region server. The Name node manages the cluster and the region servers store portions of the HBase tables and perform the work on the data.

The Protegrity HBase protector extends the functionality of the data storage framework and provides transparent data protection and unprotection using coprocessors, which provide the functionality to run code directly on region servers. The Protegrity coprocessor for HBase runs on the region servers and protects the data stored in the servers. All clients which work with HBase are supported.

The data is transparently protected or unprotected, as required, utilizing the coprocessor framework.

For more information about HBase, refer to section *HBase*.

2.4.5 Protection in Impala

Impala is an MPP SQL query engine for querying the data stored in a cluster. It provides the flexibility of the SQL format and is capable of running the queries on HDFS in HBase.

The Protegrity Impala protector extends the functionality of the Impala query engine and provides UDFs which protect or unprotect the data as it is stored or retrieved.

For more information about the Impala protector, refer to section *Impala*.

2.4.6 Protection in Spark

Spark is an execution engine that carries out batch processing of jobs in-memory and handles a wider range of computational workloads. In addition to processing a batch of stored data, Spark is capable of manipulating data in real time. You can also utilise Spark Streaming to process live data streams and store the processed data in Hadoop.

The Protegrity Spark Java protector extends the functionality of the Spark engine and provides Java APIs that protect, unprotect, or reprotect the data as it is stored or retrieved.

For more information about the Spark Java and SQL protectors, refer to section *Spark*.

The Protegrity Spark Java protector extends the functionality of the Spark engine and provides Java APIs that protect, unprotect, or reprotect the data as it is stored or retrieved.

The Protegrity Spark SQL protector provides native UDFs that can be utilized with Spark Scala to protect, unprotect, or reprotect the data as it is stored or retrieved.

You can create and submit Spark jobs using the methods listed in the following table.

Table 2-1: Creating and Submitting Spark Jobs

To create and submit Spark jobs with:	**Refer to section:**
Spark Java APIs	*Spark Java*
Spark SQL UDFs	*Spark SQL*
Spark Scala	*Spark Scala*
PySpark	*<create a section and link>*

2.5 Data Security Policy and Protection Methods

A data security policy establishes processes to ensure the security and confidentiality of sensitive information. In addition, the data security policy establishes administrative and technical safeguards against unauthorized access or use of the sensitive information.

Depending on the requirements, the data security policy typically performs the following functions:

- Classifies the data that is sensitive for the organization.
- Defines the methods to protect sensitive data, such as encryption and tokenization.
- Defines the methods to present the sensitive data, such as masking the display of sensitive information.
- Defines the access privileges of the users that would be able to access the dasta.

- Defines the time frame for privileged users to access the sensitive data.
- Enforces the security policies at the location where sensitive data is stored.
- Provides a means of auditing authorized and unauthorized accesses to the sensitive data. In addition, it can also provide a means of auditing operations to protect and unprotect the sensitive data.

The data security policy contains a number of components, such as, data elements, datastores, member sources, masks, and roles. The following list describes the functions of each of these entities:

- **Data elements** define the data protection properties for protecting sensitive data, consisting of the data securing method, data element type and its description. In addition, Data elements describe the tokenization or encryption properties, which can be associated with roles.
- **Datastores** consist of enterprise systems, which might contain the data that needs to be processed, where the policy is deployed and the data protection function is utilized.
- **Member sources** are the external sources from which users (or members) and groups of users are accessed. Examples are a file, database, LDAP, and Active Directory.
- **Masks** are a pattern of symbols and characters, that when imposed on a data field, obscures its actual value to the user. Masks effectively aid in hiding sensitive data.
- **Roles** define the levels of member access that are appropriate for various types of information. Combined with a data element, roles determine and define the unique data access privileges for each member.

Note: For more information about the data security policies, protection methods, and the data elements supported by the components of the Big Data Protector, refer to *Protection Methods Reference Guide 9.2.0.0*.

2.6 Installing and Uninstalling Big Data Protector

Note: For more information about installing and uninstalling the Big Data Protector, refer to *Installation Guide 9.2.0.0*.

2.7 Understanding the Architecture

The Log Forwarder is a log processor tool running along with the PEP server on a cluster node. It serves the purpose of collecting, aggregating, caching, and moving the logs from the PEP server (Application log) and the Big Data Protector (Audit log) to the Audit Store on the ESA and PSU.

The Log Forwarder uses the *15780* port, which is configurable, to receive Audit Log and Protection Log on each cluster node and send the logs to the Appliance running the Audit Store. The appliance can be the ESA or the Protegrity Storage Unit (PSU). The Appliance communicates with the Log Forwarder using the port *9200*.

Note:

- For more information about the Protegrity Storage Unit (PSU), refer the *Protegrity Storage Unit Guide 9.2.0.0*.
- For more information about logging, refer to the *Protegrity Log Management Guide 9.2.0.0*.
- For more information about the Audit Store, refer to the *Audit Store Guide 9.2.0.0*.

2.8 Working with the Log Forwarder

The Log Forwarder is a log processor tool running along with the PEP server on a cluster node. It serves the purpose of collecting, aggregating, caching, and moving the logs from the PEP server (Application log) and the Big Data Protector (Audit log) to the Audit Store on the ESA and PSU.

The Log Forwarder uses the *15780* port, which is configurable, to receive Audit Log and Protection Log on each cluster node and send the logs to the Appliance running the Audit Store. The appliance can be the ESA or the Protegrity Storage Unit (PSU). The Appliance communicates with the Log Forwarder using the port *9200*.

For more information about the Protegrity Storage Unit (PSU), refer the *Protegrity Storage Unit Guide 9.1.0.0*.

For more information about logging, refer to the *Protegrity Log Management Guide 9.1.0.0*.

For more information about the Audit Store, refer to the *Audit Store Guide 9.1.0.0*.

The Log Forwarder aggregates logs at 10 second intervals. For the Big Data Protector, the components of Log Forwarder are configured in the *pepserver.cfg* file.

For more information about the Log Forwarder related configurations, refer to the section *Appendix: PEP Server Configuration File* in the *Protegrity Installation Guide 9.1.0.0*.

2.8.1 Logging Architecture

Depending on the proxy configuration of the Big Data Protector, the logging architecture can be specified as following types:

- Logging Architecture of the Big Data Protector cluster without the Proxy
- Logging Architecture of the Big Data Protector cluster with the Proxy

2.8.2 Logging Architecture of the Big Data Protector Cluster without the Proxy

This section explains the logging architecture of the Big Data Protector cluster without the Proxy.

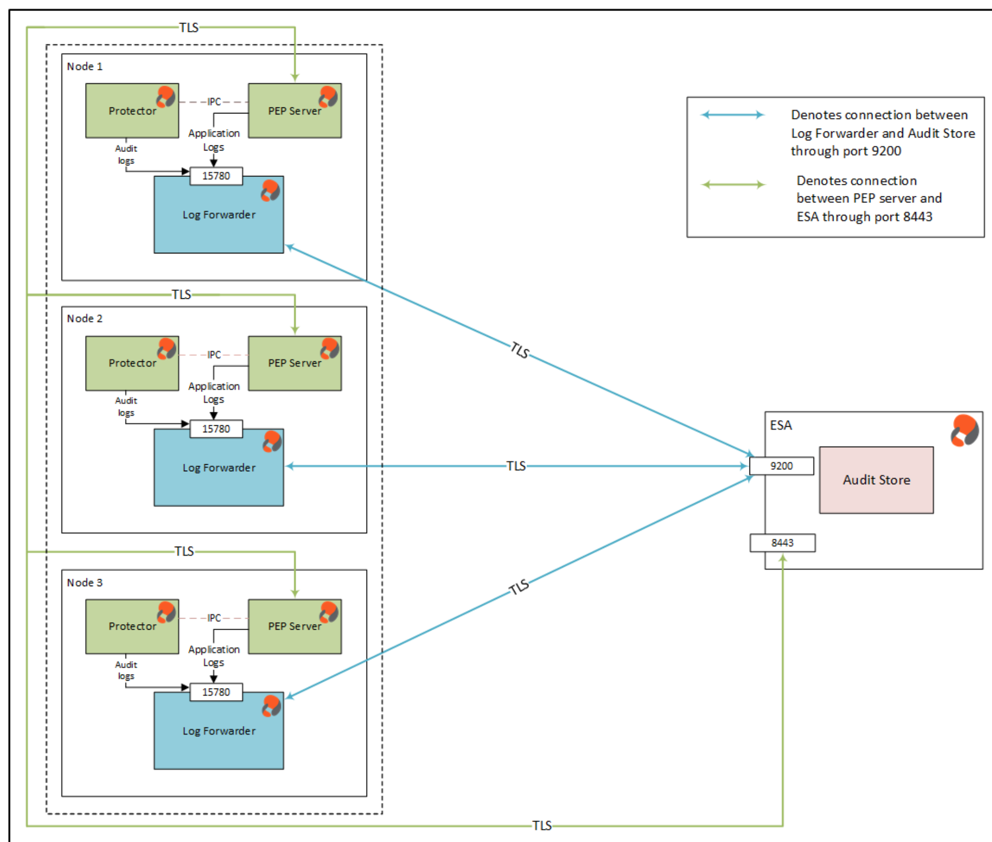

Figure 2-3: Big Data Protector Cluster Logging Architecture without the Proxy

In a multi-node architecture, the Log Forwarder, on each cluster node, collects the Application and Audit logs and the Big Data Protector and sends the logs to the to the Appliance running the Audit Store. The appliance can be the ESA or the Protegrity Storage Unit (PSU).

2.8.3 Logging Architecture of the Big Data Protector Cluster with the Proxy

This section explains the logging architecture of the Big Data Protector cluster with the Proxy.

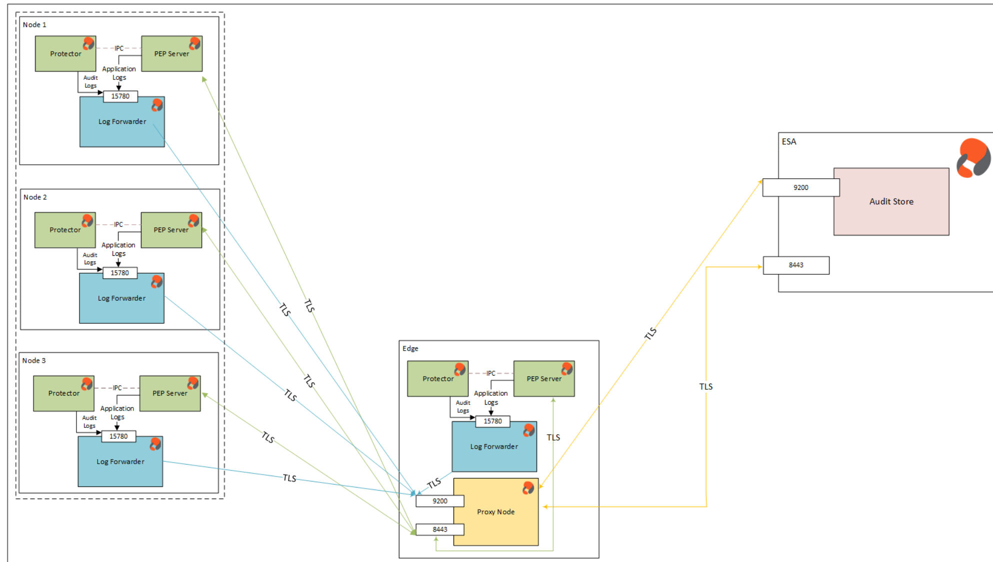

Figure 2-4: Big Data Protector Cluster Logging Architecture with the Proxy

In this scenario, the Log Forwarder, on each cluster node, collects the application and audit logs and the Big Data Protector and sends the logs to the Proxy. The Proxy sends the logs to the appliance running the Audit Store. The appliance can be the ESA or the Protegrity Storage Unit (PSU).

Chapter 3

Hadoop Application Protector

3.1 Using the Hadoop Application Protector

3.2 Prerequisites

3.3 MapReduce APIs

3.4 Sample Code Usage

3.5 Hive UDFs

3.6 Pig UDFs

3.1 Using the Hadoop Application Protector

Various jobs written in the Hadoop cluster require data fields to be stored and retrieved. This data requires protection when it is at rest. The Hadoop Application Protector provides MapReduce, Hive and Pig the power to protect data while it is being processed and stored. Application programmers using these tools can include Protegrity software in their jobs to secure data.

For more information about using the protector APIs in various Hadoop applications and samples, refer to the following sections.

3.2 Prerequisites

Ensure that the following prerequisites are met before using Hadoop Application Protector:

- The Big Data Protector is installed and configured in the Hadoop cluster.
- The security officer has created the necessary security policy which creates data elements and user roles with appropriate permissions.

  > **Note:** For more information about creating security policies, data elements and user roles, refer to *Policy Management Guide 9.2.0.0*.

- The policy is deployed across the cluster.

> **Note:** For more information about the list of all APIs available to Hadoop applications, refer to sections *MapReduce APIs*, *Hive UDFs*, and *Pig UDFs*.

3.3 MapReduce APIs

For more information about the MapReduce APIs, refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

> **Note:**
> The Protegrity MapReduce protector only supports *bytes* converted from the *string* data type.
>
> If any other data type is directly converted to *bytes* and passed as input to the API that supports *byte* as input and provides *byte* as output, then data corruption might occur.

If you are using the Bulk APIs for the MapReduce protector, then the following two modes for error handling and return codes are available:

- Default mode: Starting with the Big Data Protector, version 6.6.4, the Bulk APIs in the MapReduce protector will return the detailed error and return codes instead of *0* for *failure* and *1* for *success*. In addition, the MapReduce jobs involving Bulk APIs will provide error codes instead of throwing exceptions.

  > **Note:** For more information about the return codes for *Big Data Protector, version 9.2.0.0*, refer to section *Appendix: Return Codes*.

- Backward compatibility mode: If you need to continue using the error handling capabilities provided with Big Data Protector, version 6.6.3 or lower, that is *0* for *failure* and *1* for *success*, then you can set this mode.

3.4 Sample Code Usage

The MapReduce sample program, described in this section, is an example on how to use the Protegrity MapReduce protector APIs. The sample program utilizes the following two Java classes:

- **ProtectData.java** – This main class calls the Mapper job.
- **ProtectDataMapper.java** – This Mapper class contains the logic to fetch the input data and store the protected content as output.

3.4.1 Main Job Class – ProtectData.java

ProtectData.java

```
package com.protegrity.samples.mapreduce;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class ProtectData extends Configured implements Tool {
        @Override
        public int run(String[] args) throws Exception
        {
    //Create the Job
        Job job = new Job(getConf(), "ProtectData");
```

```
    //Set the output key and value class
       job.setOutputKeyClass(NullWritable.class);
       job.setOutputValueClass(Text.class);

    //Set the output key and value class
       job.setMapOutputKeyClass(NullWritable.class);
       job.setMapOutputValueClass(Text.class);

    //Set the Mapper class which will perform the protect job
       job.setMapperClass(ProtectDataMapper.class);

     //Set number of reducer task
       job.setNumReduceTasks( 0 );

     //Set the input and output Format class
      job.setInputFormatClass(TextInputFormat.class);
       job.setOutputFormatClass(TextOutputFormat.class);

     //Set the jar class
       job.setJarByClass(ProtectData.class);

     //Store the input path and print the input path
       Path input = new Path(args[0]);
       System.out.println(input.getName());
     //Store the output path and print the output path
       Path output = new Path(args[1]);
       System.out.println(output.getName());

     //Add input and set output path
       FileInputFormat.addInputPath(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));

     //Call the job
       return job.waitForCompletion(true) ? 0 : 1;
       }

       public static void main(String args[]) throws Exception       {
              System.exit(ToolRunner.run(new Configuration(), new ProtectData(), args));
       }  }
```

3.4.2 Mapper Class – ProtectDataMapper.java

ProtectDataMapper.java

```
package com.protegrity.samples.mapreduce;

import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
//Need to import the ptyMapReduceProtector class to use the Protegrity MapReduce protector
import com.protegrity.hadoop.mapreduce.ptyMapReduceProtector;

//Create the Mapper class i.e. ProtectDataMapper which will extends the Mapper Class
public class ProtectDataMapper extends Mapper<Object, Text, NullWritable, Text> {

    //Declare the member variable for the ptyMapReduceProtector class
    private ptyMapReduceProtector mapReduceProtector;
    //Declare the Array of Data Elements which will be required to do the protection/
unprotection
    private final String[] data_element_names = { "TOK_NAME", "TOK_PHONE", "TOK_CREDIT_CARD",
"TOK_AMOUNT" };

    //Initialize the mapreduce protector i.e ptyMapReduceProtector in the default constructor
    public ProtectDataMapper() throws Exception {
        // Create the new object for the class ptyMapReduceProtector
        mapReduceProtector = new ptyMapReduceProtector();
        // Open the session using the method " openSession("0") "
        int openSessionStatus = mapReduceProtector.openSession("0");
    }
```

```
    //Override the map method to parse the text and process it line by line
    //Split the inputs separated by delimiter "," in the line
    //Apply the protect/unprotect operation
    //Create the output text which will have protected/unprotected outputs separated by
delimiter ","
    //Write the output text to the context
    @Override
    public void map(Object key, Text value, Context context) throws IOException,
            InterruptedException
    {
        // Store the line in a variable strOneLine
        String strOneLine = value.toString();
        // Split the inputs separated by delimiter "," in the line
        StringTokenizer st = new StringTokenizer(strOneLine, ",");
        // Create the instance of StringBuilder to store the output
        StringBuilder sb = new StringBuilder();
        // Store the no of inputs in a line
        int noOfTokens = st.countTokens();
        if (mapReduceProtector != null) {
            //Iterate through the string token and apply the protect/unprotect operation
            for (int i = 0; st.hasMoreElements(); i++) {
                String data = (String)st.nextElement();
                if(i == 0) {
                    sb.append(new String(data));
                } else {
                    //To protect data, call the function protect method with parameters data
element and input data in bytes
                    //mapReduceProtector.protect( <Data Element> , <Data in bytes> )
                    //Output will be returned in bytes
                    //To unprotect data, call the function unprotect method with parameters
data element and input data in bytes
                    //mapReduceProtector.unprotect( <Data Element> , <Data in bytes> )
                    //Output will be returned in bytes
                    byte[] bResult =
                            mapReduceProtector.protect(data_element_names[i-1],
data.trim().getBytes());
                    if (bResult != null) {
                        // Store the result in string and append it to the output sb
                        sb.append(new String(bResult));
                    }
                    else {
                        // If output will be null, then store the result as "cryptoError" and
append it to the output sb
                        sb.append("cryptoError");
                    }
                }
                if(i < noOfTokens -1 ) {
                    // Append delimiter "," at the end of the processed result
                    sb.append(",");
                } } }
        // write the output text to context
        context.write(NullWritable.get(), new Text(sb.toString()));
    }

    // call flushAudits() in the cleanup method of Mapper so that all Protector
    // audit logs are flushed to Audit Store at the end of each Mapper task
    @Override
    public void cleanup(Mapper.Context context){
        mapReduceProtector.flushAudits();
    }

    //clean up the session and objects
    @Override
    protected void finalize() throws Throwable {
        //Close the session
        int closeSessionStatus = mapReduceProtector.closeSession();
        mapReduceProtector = null;
        super.finalize();
    }
}
```

3.5 Hive UDFs

For more information about the Hive User Defined Functions (UDFs), refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

Note: If you are using Ranger or Sentry, then ensure that your policy provides *create* access permissions to the required UDFs.

3.6 Pig UDFs

Note: For more information about the Pig UDFs, refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

Chapter 4

HBase

4.1 Overview of the HBase Protector
4.2 HBase Protector Usage
4.3 Adding Data Elements and Column Qualifier Mappings to a New Table
4.4 Adding Data Elements and Column Qualifier Mappings to an Existing Table
4.5 Inserting Protected Data into a Protected Table
4.6 Retrieving Protected Data from a Table
4.7 HBase Commands
4.8 Ingesting Data Securely
4.9 Extracting Data Securely

HBase is a database, which provides random read and write access to tables, consisting of rows and columns, in real-time. HBase is designed to run on commodity servers, to automatically scale as more servers are added, and is fault tolerant as data is divided across servers in the cluster. HBase tables are partitioned into multiple regions. Each region stores a range of rows in the table. Regions contain a datastore in memory and a persistent datastore (HFile). The Name node assigns multiple regions to a region server. The Name node manages the cluster and the region servers store portions of the HBase tables and perform the work on the data.

4.1 Overview of the HBase Protector

The Protegrity HBase protector extends the functionality of the data storage framework and provides transparent data protection and unprotection using coprocessors, which provide the functionality to run code directly on region servers. The Protegrity coprocessor for HBase runs on the region servers and protects the data stored in the servers. All clients which work with HBase are supported.

The data is transparently protected or unprotected, as required, utilizing the coprocessor framework.

4.2 HBase Protector Usage

The Protegrity HBase protector utilizes the *get*, *put*, and *scan* commands and calls the Protegrity coprocessor for the HBase protector. The Protegrity coprocessor for the HBase protector locates the metadata associated with the requested column qualifier and the current logged in user. If the data element is associated with the column qualifier and the current logged in user, then the HBase protector processes the data in a row based on the data elements defined by the security policy deployed in the Big Data Protector.

> **Warning:**
> The Protegrity HBase coprocessor only supports *bytes* converted from the *string* data type.

If any other data type is directly converted to *bytes* and inserted in an HBase table, which is configured with the Protegrity HBase coprocessor, then data corruption might occur.

4.3 Adding Data Elements and Column Qualifier Mappings to a New Table

In an HBase table, every column family of a table stores metadata for that family, which contain the column qualifier and data element mappings.

Users need to add metadata to the column families for defining mappings between the data element and column qualifier, when a new HBase table is created.

The following command creates a new HBase table with one column family.

```
create 'table', { NAME => 'column_family_1', METADATA => {
'DATA_ELEMENT:credit_card'=>'CC_NUMBER','DATA_ELEMENT:name'=>'TOK_CUSTOMER_NAME' } }
```

Parameters

Table 4-1: Parameters to create a new HBase Table

Parameter	Description
table	Specifies the name of the table.
column_family_1	Specifies the name of the column family.
METADATA	Specifies the data associated with the column family.
DATA_ELEMENT	Contains the column qualifier name. In the example, the column qualifier names *credit_card* and *name*, correspond to data elements *CC_NUMBER* and *TOK_CUSTOMER_NAME* respectively.

4.4 Adding Data Elements and Column Qualifier Mappings to an Existing Table

Users can add data elements and column qualifiers to an existing HBase table. Users need to alter the table to add metadata to the column families for defining mappings between the data element and column qualifier.

The following command adds data elements and column qualifier mappings to a column in an existing HBase table.

```
alter 'table', { NAME => 'column_family_1', METADATA =>
{ 'DATA_ELEMENT:credit_card'=>'CC_NUMBER', 'DATA_ELEMENT:name'=>'TOK_CUSTOMER_NAME' } }
```

Parameters

Table 4-2: Parameters to add data elements in an existing HBase Table

Parameter	Description
table	Specifies the name of the table.
column_family_1	Specifies the name of the column family.
METADATA	Specifies the data associated with the column family.
DATA_ELEMENT	Contains the column qualifier name. In the example, the column qualifier names *credit_card* and *name*, correspond to data elements *CC_NUMBER* and *TOK_CUSTOMER_NAME* respectively.

4.5 Inserting Protected Data into a Protected Table

You can ingest protected data into a protected table in HBase using the *BYPASS_COPROCESSOR* flag. If you set the *BYPASS_COPROCESSOR* flag while inserting data in the HBase table, then the Protegrity coprocessor for HBase is bypassed.

The following command bypasses the Protegrity coprocessor for HBase and ingests protected data into an HBase table.

```
put 'table', 'row_2', 'column_family:credit_card', '3603144224586181', {
ATTRIBUTES => {'BYPASS_COPROCESSOR'=>'1'}}
```

Parameters

Table 4-3: Parameters to ingest protected data into an HBase Table

Parameter	Description
table	Specifies the name of the table.
column_family	Specifies the name of the column family.
METADATA	Specifies the data associated with the column family.
ATTRIBUTES	Specifies additional parameters to consider when ingesting the protected data. In the example, the flag to bypass the Protegrity coprocessor for HBase is set.

4.6 Retrieving Protected Data from a Table

If you want to retrieve protected data from an HBase table, then you must set the *BYPASS_COPROCESSOR* flag to retrieve the data. This is required to retain the protected data as is because HBase protects and unprotects the data transparently.

The following command bypasses the Protegrity coprocessor for HBase and retrieves protected data from an HBase table.

```
scan 'table', { ATTRIBUTES => {'BYPASS_COPROCESSOR'=>'1'}}
```

Parameters

Table 4-4: Parameters to retrieve protected data from an HBase Table

Parameter	Description
table	Specifies the name of the table.
ATTRIBUTES	Specifies additional parameters to consider when ingesting the protected data. In the example, the flag to bypass the Protegrity coprocessor for HBase is set.

table: Name of the table.

ATTRIBUTES: Additional parameters to consider when ingesting the protected data. In the example, the flag to bypass the Protegrity coprocessor for HBase is set.

4.7 HBase Commands

Hadoop provides shell commands to ingest, extract, and display the data in an HBase table.

> **Note:** For more information about the commands supported by HBase, refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

4.8 Ingesting Data Securely

To ingest data into HBase securely, use the *put* command.

> **Note:** For more information about the *put* command, refer to the section *put* in *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

4.9 Extracting Data Securely

To extract data from HBase securely, use the *get* command.

> **Note:** For more information about the *get* command, refer to section *get* in the *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

Chapter 5

Impala

5.1 Overview of the Impala Protector
5.2 Impala Protector Usage
5.3 Impala UDFs
5.4 Inserting Data from a File into a Table
5.5 Protecting Existing Data
5.6 Unprotecting the Protected Data
5.7 Retrieving Data from a Table

Impala is an MPP SQL query engine for querying the data stored in a cluster. It provides the flexibility of the SQL format and is capable of running the queries on HDFS in HBase.

> **Note:** This section is applicable for the CDP-PVC-Base and CDH native installer only.

This section provides information about the Impala protector, the UDFs provided, and the commands for protecting and unprotecting data in an Impala table.

5.1 Overview of the Impala Protector

Impala is an MPP SQL query engine for querying the data stored in a cluster. The Protegrity Impala protector extends the functionality of the Impala query engine and provides UDFs which protect or unprotect the data as it is stored or retrieved.

5.2 Impala Protector Usage

The Protegrity Impala protector provides UDFs for protecting data using encryption or tokenization, and unprotecting data using decryption or detokenization.

> **Note:** Ensure that the */user/impala* path exists in HDFS with the Impala supergroup permissions.
> To verify the path, run the following command:

```
# hdfs dfs -ls /user
```

5.2.1 Creating the */user/impala* path in Impala with Supergroup Permissions

➤ To create the */user/impala* path in Impala with Supergroup permissions:

If the */user/impala* path does not exist or does not have supergroup permissions, then perform the following steps.

1. To create the */user/impala* directory in HDFS, run the following command.

```
# sudo -u hdfs dfs -mkdir /user/impala
```

2. To assign Impala supergroup permissions to the */user/impala* path, run the following command.

```
# sudo -u hdfs dfs -chown -R impala:supergroup /user/impala
```

5.3 Impala UDFs

> **Note:** For more information about the Impala UDFs, refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

5.4 Inserting Data from a File into a Table

To insert data from a file into an Impala table, ensure that the required user permissions for the directory path in HDFS are assigned for the Impala table.

5.4.1 Preparing the environment for the *basic_sample.csv* file

➤ To prepare the environment for the *basic_sample.csv* file:

1. To assign permissions to the path where data from the *basic_sample.csv* file needs to be copied, run the following command:

```
sudo -u hdfs hdfs dfs -chown root:root /tmp/basic_sample/sample/
```

2. To copy the data from the *basic_sample.csv* file into HDFS, run the following command:

```
sudo -u hdfs hdfs dfs -put basic_sample.csv /tmp/basic_sample/sample/
```

3. To verify the presence of the *basic_sample.csv* file in the HDFS path, run the following command:

```
sudo -u hdfs hdfs dfs -ls /tmp/basic_sample/sample/
```

4. To assign permissions for Impala to the path where the *basic_sample.csv* file is copied, run the following command:

```
sudo -u hdfs hdfs dfs -chown impala:supergroup /path/
```

5.4.2 Populating the table *sample_table* from the *basic_sample_data.csv* file

➤ To populate the table *sample_table* from the *basic_sample_data.csv* file:

To populate the *basic_sample* table with the data from the *basic_sample_data.csv* file, run the following query:

```
create table sample_table(colname1 colname1_format, colname2 colname2_format, colname3
colname3_format)
row format delimited fields terminated by ',';
LOAD DATA INPATH '/tmp/basic_sample/sample/' INTO TABLE sample_table;
```

where:

Table 5-1: Parameters to populate the table

Parameter	Description
sample_table	Is the name of the Impala table created to load the data from the input CSV file from the required path
colname1, colname2, colname3	Specifies the name of the columns.
colname1_format, colname2_format, colname3_format	Specifies the data types contained in the respective columns. The data types can only be any one of the following types: • *STRING* • *INT* • *DOUBLE* • *FLOAT*
ATTRIBUTES	Specifies the additional parameters to consider when ingesting the data

Note: In the example, the row format is delimited using the ‘,’ character because the row format in the input file is comma separated. If the input file is tab separated, then the the row format is delimited using '\t'.

5.5 Protecting Existing Data

To protect existing data, you must define the mapping between the columns and their respective data elements in the data security policy.

The following commands ingest cleartext data from the *basic_sample* table to the *basic_sample_protected* table in protected form using Impala UDFs.

```
create table basic_sample_protected (colname1 colname1_format, colname2 colname2_format,
colname3 colname3_format)
insert into basic_sample_protected(colname1, colname2, colname3)
select ID,pty_stringins(colname1, dataElement1),pty_stringins(colname2,
dataElement2),pty_stringins(colname3, dataElement3)
from basic_sample;
```

where:

Table 5-2: Parameters to ingest cleartext data using Impala UDFs

Parameter	Description
basic_sample_protected	Specifies the name of the table to store the protected data.
colname1, colname2, colname3	Specifies the name of the columns.
dataElement1, dataElement2, dataElement3	Specifies the data elements corresponding to the columns.
basic_sample	Specifies the name of the table containing the original data in cleartext form.

5.6 Unprotecting the Protected Data

To unprotect protected data, you must specify the following:

- The name of the table that contains the protected data
- The table that will store the unprotected data
- The columns and their respective data elements

Note: Ensure that you have the permissions to unprotect the data as required in the data security policy.

With the required permissions in place, execute the following queries to unprotect the protected data in a table and store the data in cleartext form into a different table:

```
create table table_unprotected (colname1 colname1_format, colname2 colname2_format, colname3
colname3_format)
insert into table_unprotected (colname1, colname2, colname3) select
ID,pty_stringsel(colname1, dataElement1),
pty_stringsel(colname2, dataElement2),pty_stringsel(colname3, dataElement3) from
table_protected;
```

where:

Table 5-3: Parameters to unprotect protected data

Parameter	Description
table_unprotected	Specifies the table to store the unprotected data.
colname1, colname2, colname3	Specifies the name of the columns.
dataElement1, dataElement2, dataElement3	Specifies the data elements corresponding to the columns.
table_protected	Specifies the name of the table containing protected data.

5.7 Retrieving Data from a Table

To retrieve data from a table, you must have access to the table.

To view the data contained in the table, execute the following query:

```
select * from table;
```

where:

Table 5-4: Parameters to view the data contained in a table

Parameter	Description
table	Specifies the name of the table.

Chapter 6

Spark

6.1 Overview of the Spark Protector
6.2 Spark Protector Usage
6.3 Spark Java
6.4 Spark SQL
6.5 Spark Scala

Spark is an execution engine that carries out batch processing of jobs in-memory and handles a wider range of computational workloads. In addition to processing a batch of stored data, Spark is capable of manipulating data in real time.

Spark leverages the physical memory of the Hadoop system and utilizes Resilient Distributed Datasets (RDDs) to store the data in-memory and lowers latency, if the data fits in the memory size. The data is saved on the hard drive only if required. As RDDs are the basic units of abstraction and computation in Spark, you can use the protection and unprotection APIs, provided by the Spark protector, when performing the transformation operations on an RDD.

If you need to use the Spark Protector API in a Spark Java job, then the users will have to implement the function interface as per the Spark Java programming specifications and subsequently use it in the required transformation of an RDD to tokenize the data.

This section provides information about the Spark protector, the APIs provided, and the commands for protecting and unprotecting data in a file by using the respective Spark APIs for protection or unprotection. In addition, it provides information about Spark SQL, which is a module that adds relational data processing capabilities to the Spark APIs, and a sample program for Spark Scala.

> **Note:**
> This section considers Spark, version 1.5.x, or higher as reference.

6.1 Overview of the Spark Protector

The Protegrity Spark protector extends the functionality of the Spark engine and provides APIs that protect or unprotect the data as it is stored or retrieved.

6.2 Spark Protector Usage

The Protegrity Spark protector provides APIs for protecting and reprotecting the data using encryption or tokenization, and unprotecting data by using decryption or detokenization.

> **Note:** Ensure that configure the Spark protector after installing the Big Data Protector.

6.3 Spark Java

This section describes the Spark APIs (Java) available for protection and unprotection in the Big Data Protector to build secure Big Data applications.

6.3.1 Spark Java APIs

Note: For more information about the Spark Java APIs, refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

Warning:

The Protegrity Spark protector only supports *bytes* converted from the *string* data type.

If you directly convert any other data type to *bytes* and pass that as input to the API that supports *byte* as input and provides *byte* as an output, then data corruption might occur.

6.3.2 Spark APIs and Supported Protection Methods

The following table lists the Spark APIs, the input and output data types, and the supported Protection Methods.

Note:

- Starting from the Version 7.1, Maintenance Release 1 (MR1), the DTP2 protection method is deprecated.
- For assistance in switching to a different protection method, contact Protegrity.

Table 6-1: Spark APIs and Supported Protection Methods

Operation	Input	Output	Protection Method Supported
Protect	Byte	Byte	Tokenization, Encryption, No Encyption, DTP2, CUSP
Protect	Short	Short	Tokenization, No Encyption
Protect	Short	Byte	Encryption, CUSP
Protect	Int	Int	Tokenization, No Encyption
Protect	Int	Byte	Encryption, CUSP
Protect	Long	Long	Tokenization, No Encyption
Protect	Long	Byte	Encryption, CUSP
Protect	Float	Float	Tokenization, No Encyption
Protect	Float	Byte	Encryption, CUSP
Protect	Double	Double	Tokenization, No Encyption
Protect	Double	Byte	Encryption, CUSP
Protect	String	String	Tokenization, No Encyption, DTP2
Protect	String	Byte	Encryption, CUSP
Unprotect	Byte	Byte	Tokenization, Encryption, No Encyption, DTP2, CUSP
Unprotect	Short	Short	Tokenization, NoEncyption
Unprotect	Byte	Short	Encryption, CUSP
Unprotect	Int	Int	Tokenization, No Encyption

Operation	Input	Output	Protection Method Supported
Unprotect	Byte	Int	Encryption, CUSP
Unprotect	Long	Long	Tokenization, No Encyption
Unprotect	Byte	Long	Encryption, CUSP
Unprotect	Float	Float	Tokenization, No Encyption
Unprotect	Byte	Float	Encryption, CUSP
Unprotect	Double	Double	Tokenization, No Encyption
Unprotect	Byte	Double	Encryption, CUSP
Unprotect	String	String	Tokenization, No Encyption, DTP2
Unprotect	Byte	String	Encryption, CUSP
Reprotect	Byte	Byte	Tokenization, Encryption, DTP2, CUSP **Note:** If a protected value is generated using *Byte* as both *Input* and *Output*, then only Encryption/CUSP is supported.
Reprotect	Short	Short	Tokenization
Reprotect	Int	Int	Tokenization
Reprotect	Long	Long	Tokenization
Reprotect	Float	Float	Tokenization
Reprotect	Double	Double	Tokenization
Reprotect	String	String	Tokenization, DTP2

6.3.3 Loading the Cleartext Data from a File to HDFS

You must first create a sample csv file that contains the cleartext data in comma separated value format. For example, create the *basic_sample_data.csv* file.

For more information on the sample data to be used for creating this csv file, refer to section *Sample Data*.

To load the cleartext data from the *basic_sample_data.csv* file:

To load the cleartext data from the *basic_sample_data.csv* file to HDFS, run the following command.

```
hdfs dfs fs -put <Local_Filesystem_Path>/basic_sample_data.csv <
Path_of_Cleartext_data_file>
```

where,

Parameters	Description
basic_sample_data.csv	Specifies the name of the file containing cleartext data
<Local_Filesystem_Path>	Specifies the directory path on the local machine where the *basic_sample_data.csv* file is saved.

Parameters	Description
<Path_of_Cleartext_data_file>	Specifies the HDFS directory path for the file with the cleartext data. **Note:** Ensure that the user who is running the command has read and write access to this location.

6.3.4 Protecting the Existing Data

To protect cleartext data, you must specify the name of the file, which contains the cleartext data and the name of the location that contains the file which would store the protected data.

The following command reads the cleartext data from the *basic_sample_data.csv* file and stores it in the *basic_sample_protected* directory in protected form using the Spark APIs.

```
./spark-submit --master yarn --class com.protegrity.spark.ProtectData <PROTEGRITY_DIR>/
samples/spark/lib/spark_protector_demo.jar
<Path_of_Cleartext_data_file>/basic_sample_data.csv
<Path_of_Protected_data_file>/basic_sample_protected
```

Note: Ensure that the user performing the task has the permissions to protect the data, as required, in the data security policy.

Parameters	Description
com.protegrity.spark.ProtectData	Specifies the Spark protector class for protecting the data.
spark_protector_demo.jar	Specifies the sample *.jar* file utilizing the Spark protector API for protecting data in the *.csv* file. You must create this sample *.jar* file by compiling the scala class files listed in section 9.5.1 Sample Code Usage for Spark (Scala)
<Path_of_Cleartext_data_file>	Specifies the HDFS directory path for the file with cleartext data.
<Path_of_Protected_data_file>	Specifies the HDFS directory path for the file with protected data.
basic_sample_data	Specifies the name of the file to read cleartext data.

6.3.5 Unprotecting the Protected Data

To unprotect the protected data, you must specify the name of the location that contains the file, which stores the protected data and the name of the location that contains the file to store the unprotected data.

To retrieve the protected data from the *basic_sample_protected* directory and save it in the *basic_sample_unprotected* directory in unprotected form, use the following command.

```
./spark-submit --master yarn --class com.protegrity.spark.UnProtectData <PROTEGRITY_DIR>/
samples/spark/lib/spark_protector_demo.jar
<Path_of_Protected_data_file>/basic_sample_protected_data
<Path_of_Unprotected_data_file>/basic_sample_unprotected_data
```

Note: Ensure that the user performing the task has the permissions to unprotect the data, as required, in the data security policy.

where,

Parameter	Description
com.protegrity.spark.UnProtectData	Specifies the Spark protector class for unprotecting the protected data.
spark_protector_demo.jar	Specifies the sample *.jar* file utilizing the Spark protector API for unprotecting the protected data in the *.csv* file. You must create this sample *.jar* file by compiling the scala class files listed in section 9.5.1 Sample Code Usage for Spark (Scala).
`<Path_of_Protected_data_file>/basic_sample_protected_data`	Specifies the HDFS directory path for the file with protected data.
`<Path_of_Unprotected_data_file>/basic_sample_unprotected_data`	Specifies the HDFS directory path for the file to store the unprotected data.

6.3.6 Retrieving the Unprotected Data from a File

To retrieve data from a file containing protected data, the user needs to have access to the file.

To view the unprotected data contained in the file, use the following command.

```
hadoop fs -cat <Path_of_Unprotected_data_file> /basic_sample_unprotected_data/part*
```

where,

Parameter	Description
`<Path_of_Unprotected_data_file>/basic_sample_unprotected_data`	Specifies the HDFS directory path for the file that contains the unprotected data.

6.4 Spark SQL

The Spark SQL module provides relational data processing capabilities to Spark. The module allows you to run SQL queries with Spark programs. It contains DataFrames, which is an RDD with an associated schema, that provide support for processing structured data in Hive tables.

Spark SQL enables structured data processing and programming of RDDs providing relational and procedural processing through a DataFrame API that integrates with Spark.

6.4.1 DataFrames

A DataFrame is a distributed collection of data, such as RDDs, with a corresponding schema. DataFrames can be created from a wide array of sources, such as Hive tables, external databases, structured data files, or existing RDDs.

It can act as a distributed SQL query engine and is equivalent to a table in a relational database that can be manipulated, similar to RDDs. To optimize execution, DataFrames support relational operations and track their schema.

6.4.2 SQLContext

A SQLContext is a class that is used to initialize Spark SQL. It enables applications to run SQL queries, while running SQL functions, and provides the result as a DataFrame.

HiveContext extends the functionality of SQLContext and provides capabilities to use Hive UDFs, create Hive queries, and access and modify the data in Hive tables.

The Spark SQL CLI is used to run the Hive metastore service in local mode and execute queries. When we run Spark SQL (*spark-sql*), which is client for running queries in Spark, it creates a *SparkContext* defined as *sc* and *HiveContext* defined as *sqlContext*.

6.4.3 Spark SQL UDFs

> **Note:** For more information about the Spark SQL User Defined Functions (UDFs), refer to *Protegrity APIs, UDFs, and Commands Reference Guide Release 9.2.0.0*.

> **Note:** The example code snippets provided in this section utilize SQL queries to invoke the UDFs, after they are registered, using the *sql.Context.sql()* method.

6.4.4 Inserting Data from a File into a Table

The following commands create a class named *Person* with columns to store data.

```
scala> import sqlContext.implicits._

scala> case class Person(colname1: colname1_format, colname2: colname2_format, colname3:
colname3_format)
```

The following command reads the local sample file *basic_sample_data.csv*.

```
scala> val input = sc.textFile("file:///opt/protegrity/samples/data/basic_sample_data.csv")
```

The following command creates a DataFrame by mapping the RDD to the RDD [Person] object.

```
scala> val df = input.map(x => x.split(",")).map(p => Person(p(0).toInt, p(1), p(2),
p(3))).toDF()
```

The following command registers the temporary table *sample_table*.

```
scala> df.registerTempTable("sample_table")
```

The following commands save the table *sample_table* to a Parquet file.

```
scala> import org.apache.spark.sql.SaveMode
scala> df.write.mode(SaveMode.Ignore).save("sample_table.parquet")
```

where:

Table 6-2: Parameters to insert data from a File into a Table

Parameter	Description
sample_table	Specifies the name of the table created to load the data from the input CSV file from the required path.
colname1, colname2, colname3	Specifies the name of the columns.
colname1_format, colname2_format, colname3_format	Specifies the data types contained in the respective columns.

6.4.5 Protecting Existing Data

This following command creates a Spark SQL table with the protected data.

```
"SELECT ID, " +
       "ptyProtectStr(colname1, 'dataElement1') as colname1," +
       "ptyProtectStr(colname1, 'dataElement2') as colname2," +
       "ptyProtectStr(colname3, 'dataElement3') as colname3," +
       "FROM basic_sample"
        ).registerTempTable("basic_sample_protected")
```

Note: Ensure that the user performing the task has the permissions to protect the data, as required, in the data security policy.

where:

Table 6-3: Parameters to protect the data

Parameter	Description
basic_sample_protected	Specifies the name of the table to store the protected data.
colname1, colname2, colname3	Specifies the name of the columns.
dataElement1, dataElement2, dataElement3	Specifies the data elements corresponding to the columns.
basic_sample	Specifies the name of the table containing the original data in cleartext form.
basic_sample_protected	Specifies the name of the table to store the protected data.

6.4.6 Unprotecting and Viewing the Protected Data

To unprotect and view the protected data, you need to specify the name of the table which contains the protected data, and the columns and their respective data elements.

Ensure that the user performing the task has permissions to unprotect the data as required in the data security policy.

The following commands unprotect the protected data from the table *table_protected*.

```
scala> drop table if exists table_unprotected;
scala> create table table_unprotected (colname1 colname1_format, colname2 colname2_format,
colname3 colname3_format) distributed randomly;

scala> sqlContext.sql(
      "SELECT ID," +
        "ptyUnprotectStr(colname1, 'dataElement1') as colname1," +
        "ptyUnprotectStr(colname2, 'dataElement2') as colname2," +
        "ptyUnprotectStr(colname3, 'dataElement3') as colname3," +
        "FROM table_protected"
        ).show(false)
```

Parameters

`ptyUnprotectStr`: The Protegrity Spark SQL UDF to unprotect *String* data.

`colname1, colname2, colname3`: Name of the columns.

`dataElement1, dataElement2, dataElement3`: The data elements corresponding to the columns.

`table_protected`: Table containing protected data.

6.4.7 Retrieving Data from a Table

To retrieve data from a table, the user needs to have access to the table.

The following command displays the data contained in the table.

```
scala> sqlContext.sql("SELECT * table").show()
```

Parameters

table: Name of the table.

6.4.8 Calling Spark SQL UDFs from Domain Specific Language (DSL)

You can utilize the functions of the Domain-Specific Langugage (DSL) and call Spark SQL UDFs to protect or unprotect data from the Dataframe APIs. The following sample snippet describes how to call the Spark SQL UDFs from a DSL.
Calling Spark SQL UDFs from Domain Specific Language (DSL)

```
package com.protegrity.spark.dsl

import com.protegrity.spark.PtySparkProtectorException
import org.apache.spark.sql.{Column, DataFrame, UserDefinedFunction}

/**
  * DSL API for applying protection on DataFrames implicitly.
  *
  * e.g
  * import sqlContext.implicits._
  * import com.protegrity.spark.dsl.PtySparkDSL._
  * val df = sc.parallelize(List("hello", "world")).toDF()
  * df.protect("_1", "AlphaNum")
  *    .withColumnRenamed("_1", "protected")
  *    .show()
  */
object PtySparkDSL {

  implicit class PtySparkDSL(dataFrame: DataFrame) {

    import org.apache.spark.sql.functions._

    private def applyUDFOnColumns(colname: String,
                                  dataElement: String,
                                  func: UserDefinedFunction): Seq[Column] = {
      dataFrame.schema.map { field =>
        val name = field.name
        if (name.equals(colname)) {
          func(col(colname), lit(dataElement)).as(colname)
        } else {
          column(name)
        }
      }
    }

    private def applyUDFOnColumns(colname: String, oldDataElement: String, newDataElement:
String, func: UserDefinedFunction): Seq[Column] = {
      dataFrame.schema.map { field =>
        val name = field.name
        if (name.equals(colname)) {
          func(col(colname), lit(oldDataElement), lit(newDataElement)).as(colname)
        } else {
          column(name)
        }
      }
    }

    /**
      * Returns data type of input field from DataFrame
      * @param colname
      * @return data type of the column
```

```
    */
  private def getFieldType(colname: String): String = {
    try {
      dataFrame.schema(colname).dataType.typeName
    } catch {
      case e: IllegalArgumentException =>
        throw new PtySparkProtectorException(e.getMessage)
    }
  }

  def protect(colname: String, dataElement: String): DataFrame = {
    val dataType = getFieldType(colname)
    val function = dataType match {
      case "short" => udf(com.protegrity.spark.udf.ptyProtectShort _)
      case "integer" => udf(com.protegrity.spark.udf.ptyProtectInt _)
      case "long" => udf(com.protegrity.spark.udf.ptyProtectLong _)
      case "float" => udf(com.protegrity.spark.udf.ptyProtectFloat _)
      case "double" => udf(com.protegrity.spark.udf.ptyProtectDouble _)
      case "decimal(38,18)" =>
        udf(com.protegrity.spark.udf.ptyProtectDecimal _)
      case "string" => udf(com.protegrity.spark.udf.ptyProtectStr _)
      case "date" => udf(com.protegrity.spark.udf.ptyProtectDate _)
      case "timestamp" => udf(com.protegrity.spark.udf.ptyProtectDateTime _)
      case _ =>
        throw new PtySparkProtectorException(
          "Error!! DSL API invoked on unsupported column type - " + dataType)
    }
    val columns = applyUDFOnColumns(colname, dataElement, function)
    dataFrame.select(columns: _*)
  }

  def protectUnicode(colname: String, dataElement: String): DataFrame = {
    val function = udf(com.protegrity.spark.udf.ptyProtectUnicode _)
    val columns = applyUDFOnColumns(colname, dataElement, function)
    dataFrame.select(columns: _*)
  }

  def unprotect(colname: String, dataElement: String): DataFrame = {
    val dataType = getFieldType(colname)
    val function = dataType match {
      case "short" => udf(com.protegrity.spark.udf.ptyUnprotectShort _)
      case "integer" => udf(com.protegrity.spark.udf.ptyUnprotectInt _)
      case "long" => udf(com.protegrity.spark.udf.ptyUnprotectLong _)
      case "float" => udf(com.protegrity.spark.udf.ptyUnprotectFloat _)
      case "double" => udf(com.protegrity.spark.udf.ptyUnprotectDouble _)
      case "decimal(38,18)" =>
        udf(com.protegrity.spark.udf.ptyUnprotectDecimal _)
      case "string" => udf(com.protegrity.spark.udf.ptyUnprotectStr _)
      case "date" => udf(com.protegrity.spark.udf.ptyUnprotectDate _)
      case "timestamp" =>
        udf(com.protegrity.spark.udf.ptyUnprotectDateTime _)
      case _ =>
        throw new PtySparkProtectorException(
          "Error!! DSL API invoked on unsupported column type - " + dataType)
    }
    val columns = applyUDFOnColumns(colname, dataElement, function)
    dataFrame.select(columns: _*)
  }

  def unprotectUnicode(colname: String, dataElement: String): DataFrame = {
    val function = udf(com.protegrity.spark.udf.ptyUnprotectUnicode _)
    val columns = applyUDFOnColumns(colname, dataElement, function)
    dataFrame.select(columns: _*)
  }

  def reprotect(colname: String, oldDataElement: String, newDataElement: String): DataFrame
= {
    val dataType = getFieldType(colname)
    val function = dataType match {
      case "short" => udf(com.protegrity.spark.udf.ptyReprotectShort _)
      case "integer" => udf(com.protegrity.spark.udf.ptyReprotectInt _)
      case "long" => udf(com.protegrity.spark.udf.ptyReprotectLong _)
      case "float" => udf(com.protegrity.spark.udf.ptyReprotectFloat _)
```

```
          case "double" => udf(com.protegrity.spark.udf.ptyReprotectDouble _)
          case "decimal(38,18)" =>
            udf(com.protegrity.spark.udf.ptyReprotectDecimal _)
          case "string" => udf(com.protegrity.spark.udf.ptyReprotectStr _)
          case "date" =>
            udf(com.protegrity.spark.udf.ptyReprotectDate _)
          case "timestamp" =>
            udf(com.protegrity.spark.udf.ptyReprotectDateTime _)
          case _ =>
            throw new PtySparkProtectorException(
              "Error!! DSL API invoked on unsupported column type - " + dataType)
        }
        val columns = applyUDFOnColumns(colname, oldDataElement, newDataElement, function)
        dataFrame.select(columns: _*)
      }

def reprotectUnicode(colname: String, oldDataElement: String, newDataElement: String):
DataFrame = {
  val function = udf(com.protegrity.spark.udf.ptyReprotectUnicode _)
  val columns = applyUDFOnColumns(colname, oldDataElement, newDataElement, function)
  dataFrame.select(columns: _*)
  }
  }
}
```

6.5 Spark Scala

The Protegrity Spark protector (Java) can be used with Scala to protect and reprotect the data by using encryption or tokenization, and unprotect the data by using decryption or detokenization.

> **Note:** In this Big Data Protector release, a sample code snippet for Spark Scala is provided.

6.5.1 Sample Code Usage for Spark (Scala)

The Spark protector sample program, described in this section, is an example on how to use the Protegrity Spark protector APIs with Scala.

The sample program utilizes the following three Scala classes for protecting and unprotecting data:

- **ProtectData.scala** – This main class creates the Spark context object and calls the DataLoader class for reading cleartext data.
- **UnProtectData.scala** - This main class creates the Spark Context object and calls the DataLoader class for reading protected data.
- **DataLoader.scala** - This loader class fetches the input from the input path, calls the *ProtectFunction* to protect the data, and stores the protected data as output in the output path. In addition, it fetches the input from the protected path, calls the *UnProtectFunction* to unprotect the data, and stores the cleartext content as output.

The following functions perform protection for every new line in the input or unprotection for every new line in the output.

- **ProtectFunction** - This class calls the Spark protector for every new line specified in the input to protect data.
- **UnProtectFunction** - This class calls the Spark protector for every new line specified in the input to unprotect data.

6.5.1.1 Main Job Class for Protect Operation – ProtectData.scala

ProtectData.scala

```
package com.protegrity.samples.spark.scala

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
```

```
object ProtectData {
  def main(args: Array[String]) {
    // create a SparkContext object, which tells Spark how to access a cluster.
    val sparkContext = new SparkContext(new SparkConf())
    // create the new object for class DataLoader
    val protector = new DataLoader(sparkContext)
    // Call writeProtectedData method which read clear data from input Path i.e (args[0]) and
write data in output path after protect operation
    protector.writeProtectedData(args(0), args(1), ",")
  }
}
```

6.5.1.2 Main Job Class for Unprotect Operation – UnProtectData.scala

UnProtectData.scala

```
package com.protegrity.samples.spark.scala

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object UnProtectData {
  def main(args: Array[String]) {
    val sparkContext = new SparkContext(new SparkConf())
    val protector = new DataLoader(sparkContext)
    protector.unprotectData(args(0), args(1), ",")
  }
}
```

6.5.1.3 Utility to call Protect or Unprotect Function – DataLoader.scala

DataLoader.scala

```
package com.protegrity.samples.spark.scala

import org.apache.log4j.Logger
import org.apache.spark.SparkContext

object DataLoader {
  private val logger = Logger.getLogger(classOf[DataLoader])
}
/**
 * A Data loader utility for reading & writing protected and un-protected data
 */
class DataLoader(private var sparkContext: SparkContext) {

  private var data_element_names: Array[String] = Array("TOK_NAME", "TOK_PHONE",
"TOK_CREDIT_CARD", "TOK_AMOUNT")

  private var appid: String = sparkContext.getConf.getAppId
  /**
   * Writes protected data to the output path delimited by the input delimiter
   *
   * @param inputPath - path of the input employee info file
   * @param outputPath - path where the output should be saved
   * @param delim - denotes the delimiter between the fields in the file
   */

  def writeProtectedData(inputPath: String, outputPath: String, delim: String) {
    // read lines from the input path & create RDD
    val rdd = sparkContext.textFile(inputPath)
    //import ProtectFunction
    import com.protegrity.samples.spark.scala.ProtectFunction._
    //call ProtectFunction on rdd
    rdd.ProtectFunction(delim, appid, data_element_names, outputPath)
  }

  /**
   * Reads protected data from the input path delimited by the input delimiter
   *
   * @param protectedInputPath - path of the protected employee data
```

```
   * @param unprotectedOutputPath - output path where unprotected data should be stored.
   * @param delim
   */

  def unprotectData(protectedInputPath: String, unprotectedOutputPath: String, delim: String)
{
    // read lines from the protectedInputPath & create RDD
    val protectedRdd = sparkContext.textFile(protectedInputPath)
    //import UnProtectFunction
    import com.protegrity.samples.spark.scala.UnProtectFunction._
    //call UnprotectFunction on rdd
    protectedRdd.UnprotectFunction(delim, appid, data_element_names, unprotectedOutputPath)
  }

}
```

6.5.1.4 ProtectFunction.scala

ProtectFunction.scala

```
package com.protegrity.samples.spark.scala

import java.util.ArrayList
import org.apache.spark.rdd.RDD
import com.protegrity.spark.Protector
import com.protegrity.spark.PtySparkProtector


object ProtectFunction {
  /*Defining this class as implicit,so that we can add new functionality to an RDD on the fly.
  implicits are lexically bounded i.e If we import this class, then only we can use it's
functions otherwise not*/
  implicit class Protect(rdd: RDD[String]) {
    def ProtectFunction(delim: String, appid: String, dataElement: Array[String],
protectoutputpath: String) =
      {
        val protectedRDD = rdd.map { line =>
          // splits the input seperated by delimiter in the line
          val splits = line.split(delim)
          // store first split in protectedString as we are not going to protect first split.
          var protectedString = splits(0)
          // Initialize input size
          val input = Array.ofDim[String](splits.length)
          // Initialize output size
          val output = Array.ofDim[String](splits.length)
          // Initialize errorList
          val errorList = new ArrayList[Integer]()
          // create the new object for class ptySparkProtector
          var protector: Protector = new PtySparkProtector(appid)
          // Iterate through the splits and call protect operation
          for (i <- 1 until splits.length) {
            input(i) = splits(i)
            // To protect data, call protect method with parameter dataElement, errorList,
input array and output array.output will be stored in output[]
            protector.protect(dataElement(i - 1), errorList, input, output)
            //Apppend output with protectedString
            protectedString += delim + output(i)
          }
          protectedString
        }

        // Save protectedRDD into output path
        protectedRDD.saveAsTextFile(protectoutputpath)

      }
  }
}
```

6.5.1.5 UnprotectFunction.scala

UnprotectFunction.scala

```
package com.protegrity.samples.spark.scala

import java.util.ArrayList
import org.apache.spark.rdd.RDD
import com.protegrity.spark.Protector
import com.protegrity.spark.PtySparkProtector


object UnProtectFunction {
  /*Defining this class as implicit,so that we can add new functionality to an RDD on the fly.
  implicits are lexically bounded i.e If we import this class, then only we can use it's
functions otherwise not*/
  implicit class Unprotect(protectedRDD: RDD[String]) {
    def UnprotectFunction(delim: String, appid: String, dataElement: Array[String],
unprotectoutputpath: String) =
      {
        val unprotectedRDD = protectedRDD.map { line =>
          // splits the input seperated by delimiter in the line
          val splits = line.split(delim)
          // store first split in unprotectedString
          var unprotectedString = splits(0)
          // Initialize input size
          val input = Array.ofDim[String](splits.length)
          // Initialize output size
          val output = Array.ofDim[String](splits.length)
          // Initialize errorList
          val errorList = new ArrayList[Integer]()
          // create the object for class ptySparkProtector
          var protector: Protector = new PtySparkProtector(appid)
          // Iterate through the splits and call unprotect operation
          for (i <- 1 until splits.length) {
            input(i) = splits(i)
            // To unprotect data, call unprotect method with parameter dataElement,
errorList, input array and output array.output will be stored in output[]
            protector.unprotect(dataElement(i - 1), errorList, input, output)
            //Apppend output with protectedString
            unprotectedString += delim + output(i)
          }
          unprotectedString
        }

        // Save unprotectedRDD into output path
        unprotectedRDD.saveAsTextFile(unprotectoutputpath)
      }
  }
}
```

Chapter 7

Appendix: Return Codes

If you are using MapReduce, Hive, Pig, HBase, or Spark, and any failures occur, then the protector throws an exception. The exception consists of an error code and error message. The following table lists all possible error codes and error descriptions.

The following table lists all possible return codes provided to the Core log files.

Table 7-1: Core Log Return Codes

Code	Error	Error Description
0	NONE	
1	USER_NOT_FOUND	The user name could not be found in the policy.
2	DATA_ELEMENT_NOT_FOUND	The data element could not be found in the policy.
3	PERMISSION_DENIED	The user does not have the required permissions to perform the requested operation.
4	TIME_PERMISSION_DENIED	The user does not have the appropriate permissions to perform the requested operation at this point in time.
5	INTEGRITY_CHECK_FAILED	Integrity check failed.
6	PROTECT_SUCCESS	The operation to protect the data was successful.
7	PROTECT_FAILED	The operation to protect the data failed.
8	UNPROTECT_SUCCESS	The operation to unprotect the data was successful.
9	UNPROTECT_FAILED	The operation to unprotect the data failed.
10	OK_ACCESS	The user has the required permissions to perform the requested operation. This return code ensures a verification and no data is protected or unprotected.
11	INACTIVE_KEYID_USED	The operation to unprotect the data was successful using an inactive Key ID.
12	INVALID_PARAM	The input is null or not within allowed limits.
13	INTERNAL_ERROR	An internal error occurring in a function call after the Provider is started.
14	LOAD_KEY_FAILED	Failed to load the data encryption key.
17	INIT_FAILED	The PEP server failed to initialize, which is a fatal error.
20	OUT_OF_MEMORY	Failed to allocate memory.
21	BUFFER_TOO_SMALL	The input or output buffer is very small.
22	INPUT_TOO_SHORT	The data is too short to be protected or unprotected.
23	INPUT_TOO_LONG	The data is too long to be protected or unprotected.

Code	Error	Error Description
25	USERNAME_TOO_LONG	The user name is longer than the maximum supported length of the user name that can be used for protect or unprotect operations.
26	UNSUPPORTED	The algorithm or action for the specific data element is unsupported.
27	APPLICATION_AUTHORIZED	The application is authorized.
28	APPLICATION_NOT_AUTHORIZED	The application is not authorized.
31	EMPTY_POLICY	The policy is empty.
32	DELETE_SUCCESS	The operation to delete the data was successful.
33	DELETE_FAILED	The operation to delete the data failed.
34	CREATE_SUCCESS	The operation to create or add the data was successful.
35	CREATE_FAILED	The operation to create or add the data failed.
36	MNGPROT_SUCCESS	The management of the protection operation was successful.
37	MNGPROT_FAILED	The management of the protection operation failed.
40	LICENSE_EXPIRED	The license is not valid or the current date is beyond the license expiration date.
41	METHOD_RESTRICTED	The use of the Protection method is restricted by license.
42	LICENSE_INVALID	The license is invalid or the time is prior to the start of the license tenure.
44	INVALID_FORMAT	The content of the input data is invalid.
46	INVALID_POLICY	It is used for a z/OS Query regarding the default data element when the policy name is not found.
49	UNSUPPORTED_ENCODING	The input encoding for the specific data element is not supported.
50	REPROTECT_SUCCESS	The data reprotection was successful.
51	LOG_UNREACHABLE	The logs cannot be sent because the log framework is not accessible.

The following table lists all possible result codes provided as a result of operations performed on the Core.

Table 7-2: Core Result Codes

Code	Error	Error Description
1	SUCCESS	The operation was successful.
0	FAILED	The operation failed.
-1	INVALID_PARAMETER	The parameter is invalid.
-2	EOF	The end of file was reached.
-3	BUSY	The operation is already in progress or the PEP server is busy with some other operation.
-4	TIMEOUT	The time-out threshold was reached as the PEP server was waiting for a response.
-5	ALREADY_EXISTS	The object, such as file, already exists.
-6	ACCESS_DENIED	The permission to access the object was denied.
-7	PARSE_ERROR	The error occurred when the contents were parsed.
-8	NOT_FOUND	The search operation was not successful.

Code	Error	Error Description
-9	NOT_SUPPORTED	The operation is not supported.
-10	CONNECTION_REFUSED	The connection was refused.
-11	DISCONNECTED	The connection was terminated.
-12	UNREACHABLE	The Internet link is down or the host is not reachable.
-13	ADDRESS_IN_USE	The IP Adddress or port is already utilized.
-14	OUT_OF_MEMORY	The operation to allocate memory failed.
-15	CRC_ERROR	The CRC check failed.
-16	BUFFER_TOO_SMALL	The buffer size is very small.
-17	BAD_REQUEST	The message received was not in a standard format.
-18	INVALID_STRING_LENGTH	The input string is very long.
-19	INVALID_TYPE	The incorrect type of <NEED INPUTS> was used.
-20	READONLY_OBJECT	The object is set with read-only access.
-21	SERVICE_FAILED	The service failed.
-22	ALREADY_CONNECTED	The Administrator is already connected to the server.
-23	INVALID_KEY	The key is invalid.
-24	INTEGRITY_ERROR	The integrity check failed.
-25	LOGIN_FAILED	The attempt to login failed.
-26	NOT_AVAILABLE	The object is not available.
-27	NOT_EXIST	The object does not exist.
-28	SET_FAILED	The Set operation failed.
-29	GET_FAILED	The Get operation failed.
-30	READ_FAILED	The Read operation failed.
-31	WRITE_FAILED	The Write operation failed.
-33	REWRITE_FAILED	The Rewrite operation failed.
-34	DELETE_FAILED	The Delete operation failed.
-35	UPDATE_FAILED	The Update operation failed.
-36	SIGN_FAILED	The Sign operation failed.
-37	VERIFY_FAILED	The Verification failed.
-38	ENCRYPT_FAILED	The Encrypt operation failed.
-39	DECRYPT_FAILED	The Decrypt operation failed.
-40	REENCRYPT_FAILED	The Reencrypt operation failed.
-41	EXPIRED	The object has expired.
-42	REVOKED	The object has been revoked.
-43	INVALID_FORMAT	The format is invalid.
-44	HASH_FAILED	The Hash operation failed.
-45	NOT_DEFINED	The property or setting is not defined.
-46	NOT_INITIALIZED	The service requested or function is performed on an object that is not initialized.
-47	POLICY_LOCKED	The Policy is locked.
-48	THROW_EXCEPTION	The error message is used to convey that an exception should be thrown during decryption.

Code	Error	Error Description
-49	USER_AUTHENTICATION_FAILED	The Authentication operation failed.
-54	INVALID_CARD_TYPE	The credit card number provided does not confirm to the required credit card format.
-55	LICENSE_AUDITONLY	The License provided is for the audit functionality and only *No Encryption* data elements are allowed.
-56	NO_VALID_CIPHERS	No valid ciphers were found.
-57	NO_VALID_PROTOCOLS	No valid protocols were found.
-201	CRYPT_KEY_DATA_ILLEGAL	The key data specified is invalid.
-202	CRYPT_INTEGRITY_ERROR	The integrity check for the data failed.
-203	CRYPT_DATA_LEN_ILLEGAL	The data length specified is invalid.
-204	CRYPT_LOGIN_FAILURE	The Crypto login failed.
-205	CRYPT_CONTEXT_IN_USE	An attempt to close a key being used is made.
-206	CRYPT_NO_TOKEN	The hardware token is available.
-207	CRYPT_OBJECT_EXISTS	The object to be created already exists.
-208	CRYPT_OBJECT_MISSING	A request for a non-existing object is made.
-221	X509_SET_DATA	The operation to set data in the object failed.
-222	X509_GET_DATA	The operation to get data from the object failed.
-223	X509_SIGN_OBJECT	The operation to sign the object failed.
-224	X509_VERIFY_OBJECT	The verification operation for the object failed.
-231	SSL_CERT_EXPIRED	The certificate has expired.
-232	SSL_CERT_REVOKED	The certificate has been revoked.
-233	SSL_CERT_UNKNOWN	The Trusted certificate was not found.
-234	SSL_CERT_VERIFY_FAILED	The certificate cound not be verified.
-235	SSL_FAILED	A general SSL error occurs.
-241	KEY_ID_FORMAT_ERROR	The format on the Key ID is invalid.
-242	KEY_CLASS_FORMAT_ERROR	The format on the KeyClass is invalid.
-243	KEY_EXPIRED	The key expired.
-250	FIPS_MODE_FAILED	The FIPS mode failed.

Appendix A

Appendix: Migrating Tokenized Unicode Data from and to a Teradata Database

8.1 Migrating Tokenized Unicode Data from a Teradata Database
8.2 Migrating Tokenized Unicode Data to a Teradata Database

This section describes the procedures for migrating tokenized Unicode data from and to a Teradata database.

> **Note:** This section is only applicable for Legacy Unicode and Base64 Unicode data element.

> **Note:** This section considers the Teradata database for reference.

> **Note:** In addition to the Teradata database, the Big Data Protector works with other databases, such as Netezza, Greenplum, and so on.

8.1 Migrating Tokenized Unicode Data from a Teradata Database

This section describes the task to unprotect the tokenized Unicode data in Hive, Impala, or Spark, which was tokenized in the Teradata database using the Protegrity Database Protector and then migrated to Hive, Impala, MapReduce, or Spark.

> **Note:**
> Ensure that the data elements used in the data security policy, deployed on the Teradata Database Protector and Big Data Protector machines are uniform.

8.1.1 Migrating Tokenized Unicode data from Teradata database to Hive or Impala and unprotecting it using Hive or Impala protector

➤ To migrate Tokenized Unicode data from Teradata database to Hive or Impala and unprotect it using Hive or Impala protector:

1. Tokenize the Unicode data in the Teradata database using Protegrity Database Protector.
2. Migrate the tokenized Unicode data from the Teradata database to Hive or Impala.
3. To unprotect the tokenized Unicode data on Hive or Impala, ensure that the following UDFs are used, as required:
   - Hive: *ptyUnprotectUnicode()*
   - Impala: *pty_UnicodeStringSel()*

8.1.2 Migrating Tokenized Unicode data from a Teradata database to Hadoop and Unprotecting it using MapReduce or Spark protector

To migrate Tokenized Unicode data from a Teradata database to Hadoop and unprotect it using MapReduce or Spark protector:

1. Migrate the tokenized Unicode data to the Hadoop ecosystem using any data migration utilities.
2. To unprotect the tokenized Unicode data using MapReduce or Spark, ensure that the following APIs are used, as required:
   - MapReduce: *public byte[] unprotect(String dataElement, byte[] data)*
   - Spark: *void unprotect(String dataElement, List<Integer> errorIndex, byte[][] input, byte[][] output)*
3. Convert the protected tokens to bytes using UTF-8 encoding.
4. Send the data as input to the Unprotect API in the MapReduce or Spark protector, as required.
5. Convert the unprotected output in bytes to *String* using UTF-16LE encoding.

   The *string* data will display the data in cleartext format.

   The following sample code snippet describes how to unprotect the Tokenized Unicode data, that is migrated from a Teradata database to Hadoop, using the MapReduce or Spark protector.

```
private Protector protector = null;
String[] unprotectinput= new String[SIZE] ;
byte[][] inputValueByte = new byte [unprotectinput.length][];
StringBuilder unprotectedString = new StringBuilder();
int x=0;
for (x=0; x< unprotectinput.length; x++)
inputValueByte[x]= unprotectinput[x].getBytes(StandardCharsets.UTF_8);  // Point a
implementation
protector.unprotect(DATAELEMENT_NAME, errorIndexList, inputValueByte,
outputValueByte); // Point b implementation
unprotectedString.append(new String(outputValueByte[j],StandardCharsets.UTF_16LE))//
Point c implementation
```

8.2 Migrating Tokenized Unicode Data to a Teradata Database

This section describes the task to protect Unicode data in Hive, Impala, MapReduce, or Spark, migrate it to a Teradata database, and then unprotect the tokenized Unicode data using the Protegrity Database Protector.

> **Note:**
> Ensure that the data elements used in the data security policy, deployed on the Teradata Database Protector and Big Data Protector machines are uniform.

8.2.1 Migrating Tokenized Unicode data using Hive or Impala protector to Teradata database

➤ To migrate Tokenized Unicode data using Hive or Impala protector to Teradata database:

1. To protect the Unicode data on Hive or Impala, ensure that the following UDFs are used, as required:
   - Hive: *ptyProtectUnicode()*
   - Impala: *pty_UnicodeStringIns()*
2. Migrate the tokenized Unicode data from Hive or Impala to the Teradata database.
3. To unprotect the tokenized Unicode data in the Teradata database, use the Protegrity Database Protector.

8.2.2 Protecting Unicode data using MapReduce or Spark protector and Migrating it to a Teradata database

➤ To protect Unicode data using MapReduce or Spark protector and migrate it to a Teradata database:

1. Convert the cleartext format Unicode data to bytes using UTF-16LE encoding.
2. To migrate the tokenized Unicode data using MapReduce or Spark to the Teradata database, ensure that the following APIs are used, as required:
   - MapReduce: *public byte[] protect(String dataElement, byte[] data)*
   - Spark: *void protect(String dataElement, List<Integer> errorIndex, byte[][] input, byte[][] output)*
3. Send the data as input to the Protect API in the MapReduce or Spark protector, as required.
4. Convert the protected output in bytes to *String* using UTF-8 encoding.

   The output is protected tokenized data.
5. Migrate the protected data to the Teradata database using any data migration utilities.

   The following sample code snippet describes how to protect Unicode data using the MapReduce or Spark protector, and migrating it to a Teradata database.

```
private Protector protector = null;
String[] clear_data = new String[SIZE] ;
byte[][] inputValueByte = new byte [clear_data.length][];
StringBuilder protectedString = new StringBuilder();
inputValueByte= data.getBytes(StandardCharsets.UTF_16LE); //Point a implementation
protector.protect(DATAELEMENT_NAME, errorIndexList, inputValueByte, outputValueByte); //
Point b implementation
int x=0;
for (x=0; x<outputValueByte.length; x++)
protectedString.append(new String(outputValueByte[x],StandardCharsets.UTF_8)); //Point c
implementation
```