HAWQ TDE Design

Name	Date	Reason For Changes	Version	Status
Hongxu Ma, Amy Bai, Ivan Weng	2016-12-07	Initial version	v0.1	Draft
Ivan Weng, Amy Bai	2016-12-26	Revision based on feedback	v0.2	Draft

1 Target **2**

2 Background **2**

2.1 What is TDE(Transparent Data Encryption) 2

2.2 What is libhdfs3 2

2.3 Key concepts of TDE 3

2.3.1 Encryption Zone and Encryption Zone Key 3

2.3.2 Data Encryption Key and Encrypted Data Encryption Key 3

2.3.3 Hadoop key management server 3

2.4 Write process of client with TDE 4

2.5 Read process of client with TDE 6

3 Implementation **7**

3.1 User Interface 7

3.1.1 Which level should be encrypted 7

3.1.2 Configuration File 7

3.1.3 GUC values for TDE 8

3.2 Overview of libhdfs3 read/write process 8

3.2.1 Write process of libhdfs3 8

3.2.2 Read process of libhdfs3 9

3.3 How to communicate with KMS 10

3.4 Security 11

3.5 Impact to other components 11

3.5.1 Register 11

3.5.2 InputFormat 11

3.6 Upgrade 12

4 Test Plan **12**

5 Reference **13**

1 Target

TDE (Transparent Data Encryption) in Hadoop HDFS provides transparent and end-to-end data encryption for data. The support of TDE in Apache HAWQ make sure user data, especially important data, can be secured during data access. Though there is performance penalty when it is enabled.

Current libhdfs3 does not support HDFS data access with TDE. For instance, Apache HAWQ cannot read and write data correctly with current libhdfs3 library. It can only get encrypted data as shown in Figure 1-1.

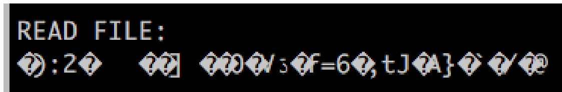

Figure 1-1. Accessing encrypted data through current libhdfs3

The support of TDE in Apache HAWQ is implemented by leveraging existing HDFS access library libhdfs3. The design is to enhance libhdfs3 so that HDFS data can be encrypted and decrypted in transit and at rest.

The plan is to encrypt global hdfs data, including filespace, tablespace, database, and table in Apache HAWQ.

2 Background

2.1 What is TDE(Transparent Data Encryption)

HDFS implements transparent, end-to-end encryption. Once configured, data read from and written to special HDFS directories is transparently encrypted and decrypted without requiring changes to user application code. This encryption is also end-to-end, which means the data can only be encrypted and decrypted by the client. HDFS never stores or has access to unencrypted data or unencrypted data encryption keys. This satisfies two typical requirements for encryption: at-rest encryption (meaning data on persistent media, such as a disk) as well as in-transit encryption (e.g. when data is travelling over the network).
(Reference 6.2 for more details)

2.2 What is libhdfs3

libhdfs3 is a c/c++ version native HDFS client. It is a key library in Apache HAWQ providing full featured HDFS filesystem manipulation and high performance data access. Therefore, it is necessary for libhdfs3 to catch new features introduced by Apache Hadoop HDFS.

For more details, please check
https://github.com/Pivotal-Data-Attic/pivotalrd-libhdfs3

2.3 Key concepts of TDE

2.3.1 Encryption Zone and Encryption Zone Key

Encryption Zone (EZ) is an HDFS directory, all files in it are encrypted. Encryption zone has an Encryption Zone Key (EZK) associated. One EZK serves as the master key for all files in the corresponding encryption zone.

2.3.2 Data Encryption Key and Encrypted Data Encryption Key

The key to encrypt the file is referred to as a Data Encryption Key (DEK). When storing a file inside an encrypted zone, the NameNode asks the Hadoop Key Management Server (KMS) to generate a new Encrypted Data Encryption Key (EDEK) encrypted with the EZK. The EDEK is stored as part of that file's metadata on the NameNode.

When reading a file stored in an encryption zone, the NameNode provides the client with the file's EDEK and the encryption zone key version used to encrypt the EDEK. The client then asks the KMS to decrypt the EDEK, which involves checking that the client has permission to access the encryption zone key version. Assuming that is successful, the client uses the DEK to decrypt the file's contents.

The client has to decrypt the encrypted key stored in the metadata, and then use the key to decrypt the the blocks.

2.3.3 Hadoop key management server

All EZKs are stored in a keystore and Hadoop Key Management Server (KMS) accesses the keys from the keystore using KeyProvider implementations. Key security guarantees that the keys and HDFS permissions are completely separated. Reference 6.1 for more details.

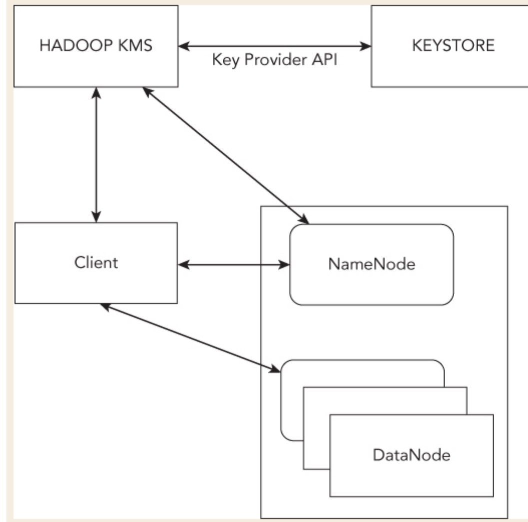

Figure 2-1. KMS co-works with HDFS components

2.4 Write process of client with TDE

When storing a file inside an encrypted zone, a key is generated and the data is encrypted using the key. For each file, a new key is generated and the encrypted key is stored as part of the file's metadata on the NameNode. The key to encrypt the file is referred to as a Data Encryption Key (DEK). The Write process of client with TDE as shown in Figure 2-2.
The key part need to be implemented at client to support HDFS write with TDE is highlighted as green.

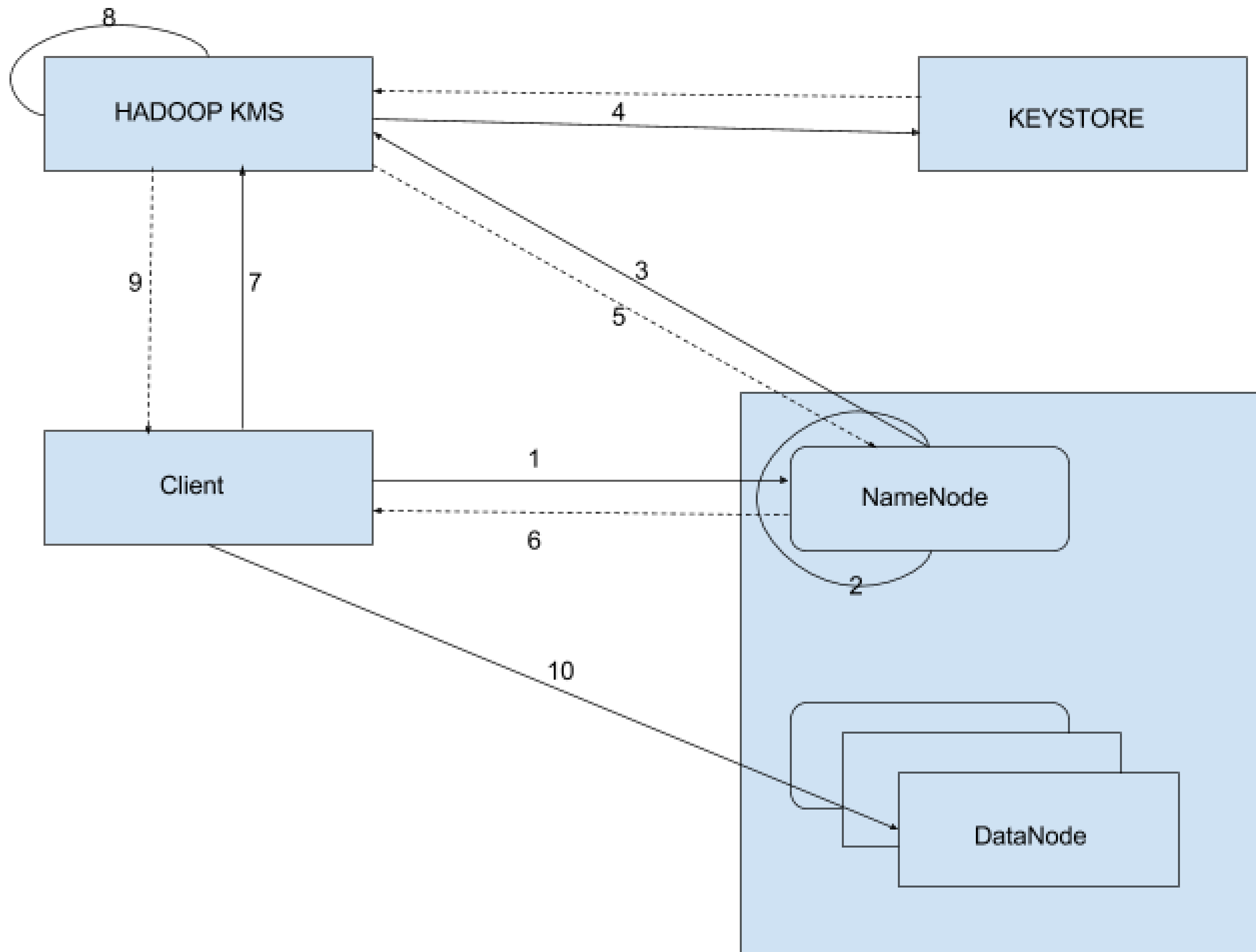

Figure 2-2. Write process of TDE

1. *The client issues a command to store a new file under /path/to/dataset.*
2. *The NameNode checks if the user has access to create a file under the specified path based on file permissions and ACLs.*
3. *The NameNode requests Hadoop KMS to create a new key (DEK) and also provide the EZK namely master_key.*
4. *The Hadoop KMS generates a new key, DEK.The Hadoop KMS retrieves the EZK (master_key) from the key store and encrypts the DEK using master_key to generate the EDEK.*
5. *The Hadoop KMS provides the EDEK to the NameNode and NameNode persists the EDEK as an extended attribute for the file metadata.*
6. *The NameNode provides the EDEK to the HDFS client.*
7. *The HDFS client sends the EDEK to the Hadoop KMS, requesting the DEK.*
8. *The Hadoop KMS checks if the user running the HDFS client has access to the EZK. Note that this authorization check is different from file permissions or ACLs.*
9. *If the user has permissions, then the Hadoop KMS decrypts the EDEK using the EZK and provides the DEK to the HDFS client.*

10. *The HDFS client encrypts data using the DEK and writes the encrypted data blocks to HDFS.*

For more details, please check:
http://www.wrox.com/WileyCDA/WroxTitle/Professional-Hadoop.productCd-111926717X.html

2.5 Read process of client with TDE

When reading a file stored in an encryption zone, the client needs to decrypt the encrypted key stored in the metadata file, and then use the key to decrypt the content of the blocks. The Read process of client with TDE as shown in Figure 2-3 .
The key part need to be implemented at client to support HDFS write with TDE is highlighted as green.

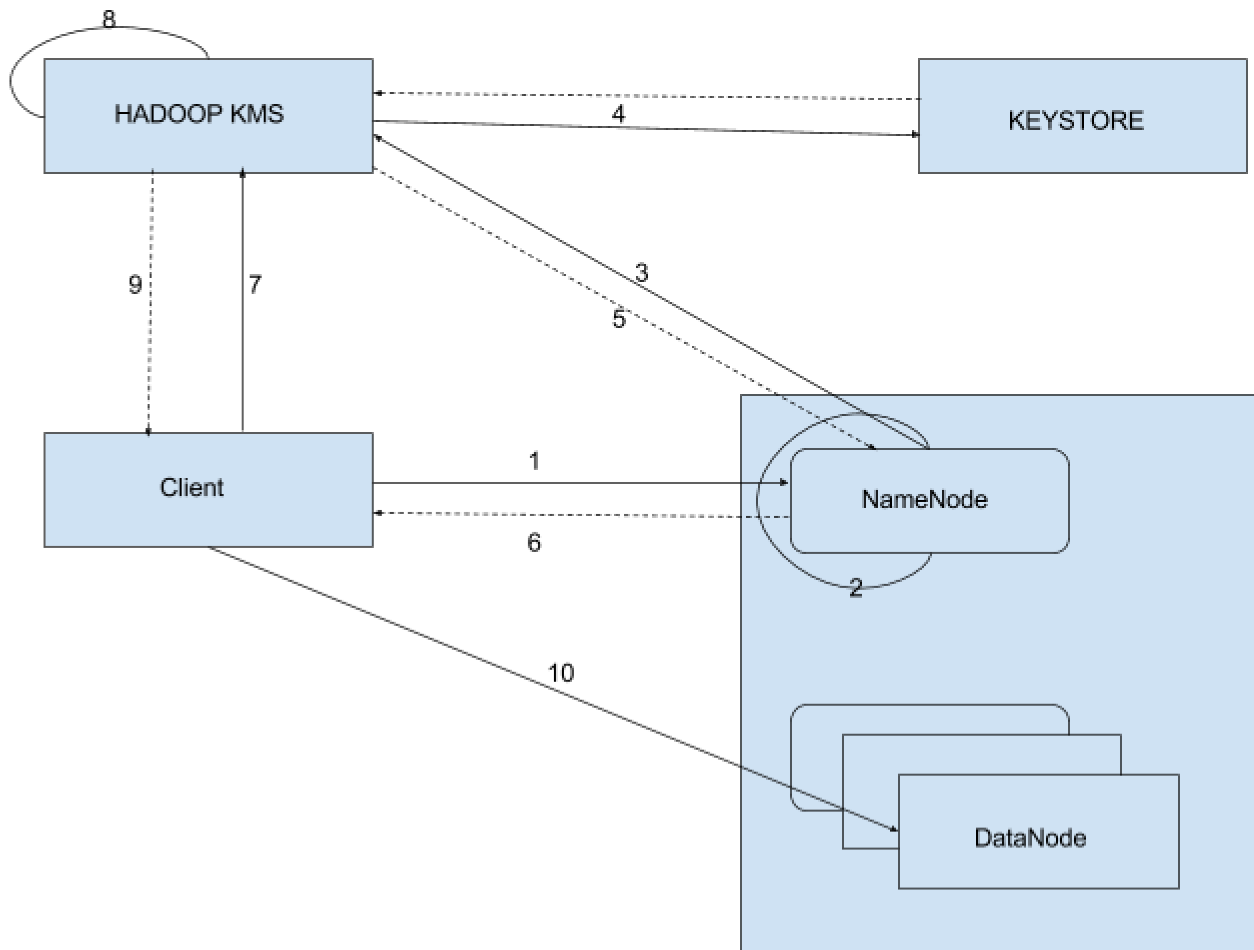

Figure 2-3. Read process of TDE

1. *The client invokes a command to read the file. Steps of 2-9 are same as the "2.4 Write process of client with TDE".*

10. *The HDFS client reads the encrypted data blocks from DataNodes, and decrypts them with the DEK.*

For more details, please check:
http://www.wrox.com/WileyCDA/WroxTitle/Professional-Hadoop.productCd-111926717X.html

3 Implementation

3.1 User Interface

3.1.1 Which level should be encrypted

TDE support in Apache HAWQ can be at five levels: 1) global level; 2) filespace level; 3) tablespace level; 4) database level; and 5) table level, descending in granularity from grosser to finer.

The finer the granularity is, the more flexible for user to control data encryption. In the meanwhile, it introduces overhead for user to define data encryption option on different database objects. For example, if the encryption can be at table level, user need to add option in create table statement to specify if the data in that need to be encrypted or not. Furthermore, it increases complexity of design and implementation regarding TDE support in Apache HAWQ.

It is planned to encrypt hawq global hdfs data, including filespace, tablespace, database, and table, in initial version of TDE support in Apache HAWQ. This will reduce user overhead at a very large extent as well as make it transparent at most.

Fine-grained support of TDE in Apache HAWQ, i.e., table level, will be considered in the future.

3.1.2 Configuration File

Configuration File	Configuration Key	Configuration Value Example
hdfs-site.xml	dfs.encryption.key.provider.uri	KMS server address <property> <name>dfs.encryption.key.provider.uri</name>

		<value>kms://http@localhost:16000/kms</value> <description>kms server</description> </property>
core-site.xml	hadoop.security.key.provider.path	<property> <name>hadoop.security.key.provider.path</name> <value>kms://http@localhost:16000/kms</value> </property>
	hadoop.security.crypto.buffer.size	Crypto buffer size. Default: 8192

For more optional configurations , please refer to :
http://www.cloudera.com/content/www/zh-CN/documentation/enterprise/5-3-x/topics/cdh_sg_hdfs_encryption.html

3.1.3 GUC values for TDE

In order to make TDE easily controlled by user, we make a guc value for user. If user would like to enable TDE function , user just needs to set guc value of enable_hdfs_tde to true before HAWQ initialization. On the contrary, user can set enable_hdfs_tde to false if user don’t need the TDE function.

3.2 Overview of libhdfs3 read/write process

3.2.1 Write process of libhdfs3

The write process of libhdfs3 is shown in Figure 3-1. We can clearly see the difference between Figure 3-1 and the write process of TDE above in Figure 2-1.

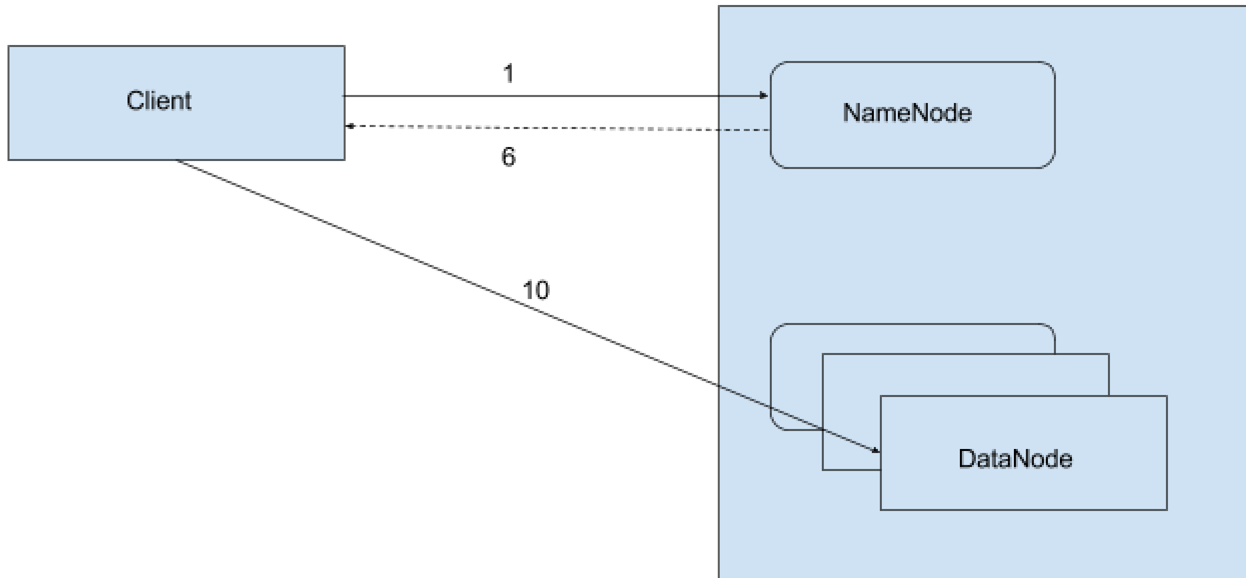

Figure 3-1. Write process of libhdfs3

1. HDFS Client issues a command to store a new file , then HDFS Client sends append RPC to NameNode.

6. NameNode responses append RPC to HDFS Client.

10. HDFS Client writes data to DataNode.

According to Figure 2-1, there are several parts we should do to implement TDE:

For step 6, NameNode provides EDEK to HDFS Client should be added.

For step 7, HDFS Client sends EDEK to Hadoop KMS, requesting DEK should be added.

For step 9, Hadoop KMS provides DEK to HDFS Client should be added.

For step 10, HDFS Client encrypts data using DEK and writes the encrypted data blocks to HDFS should be added.

3.2.2 Read process of libhdfs3

The read process of libhdfs3 is shown in Figure 3-2. We can clearly see the difference between Figure 3-2 and the read process of TDE above in Figure 2-2.

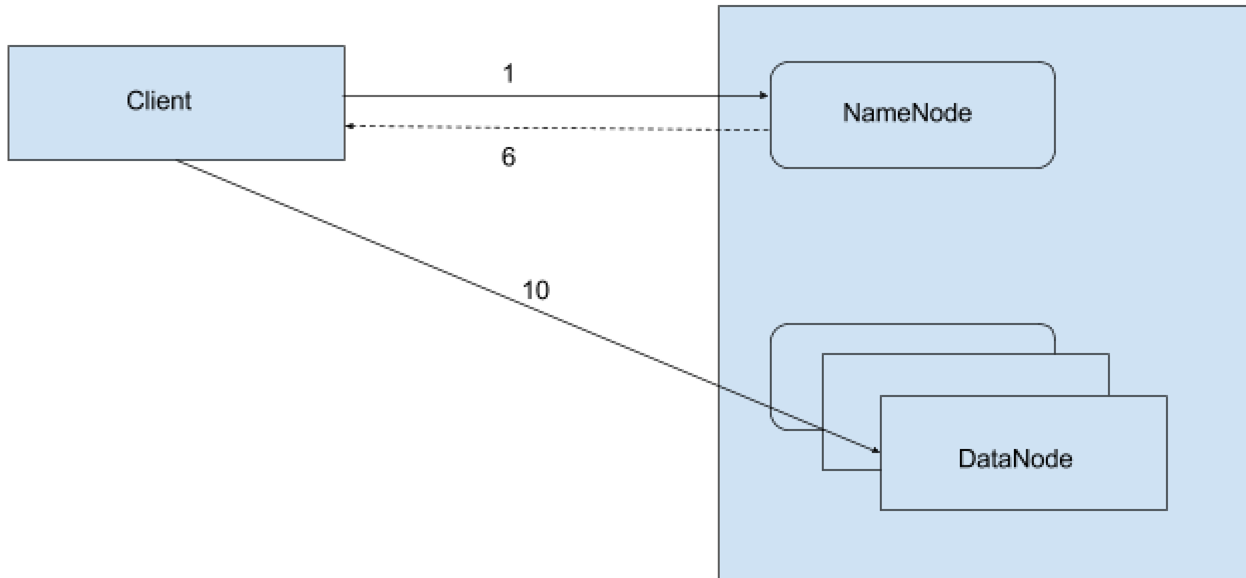

Figure 3-2. Read process of libhdfs3

1. HDFS Client issues a command to read the file , then HDFS Client sends read RPC to NameNode.

6. NameNode responses read RPC to HDFS Client.

10. HDFS Client reads the data blocks from DataNodes.

According to Figure 2-2, there are several parts we should do to implement TDE:

For step 6, NameNode provides EDEK to HDFS Client should be added.

For step 7, HDFS Client sends EDEK to Hadoop KMS, requesting the DEK should be added.

For step 9, Hadoop KMS provides the DEK to HDFS client should be added.

For step 10, HDFS Client reads the encrypted data blocks from DataNodes, and decrypts the data blocks should be added.

3.3 How to communicate with KMS

Client needs the decrypt api to communicate with KMS as Figure 3-2 shown which is http restful api.

Decrypt Encrypted Key

REQUEST:

```
POST http://HOST:PORT/kms/v1/keyversion/<version-name>/_eek?ee_op=decrypt
Content-Type: application/json

{
  "name"          : "<key-name>",
  "iv"            : "<iv>",          //base64
  "material"      : "<material>",    //base64
}
```

RESPONSE:

```
200 OK
Content-Type: application/json

{
  "name"          : "EK",
  "material"      : "<material>",    //base64
}
```

Figure 3-3. KMS decrypt API

3.4 Security

About KMS ACL, as for now, libhdfs3 does not distinguish user access to hadoop, so for TDE, KMS ACL will be the same for all the users, we will ignore this.

About SSL transmission security between client and KMS, TDE will do the same as hadoop libhdfs client.

3.5 Impact to other components

3.5.1 Register

If it enable TDE in hawq cluster, hawq register is not supported.
If it disable TDE in hawq cluster, it only support register non-encrypted hdfs data into hawq table. It cannot register encrypted hdfs data.

3.5.2 InputFormat

TDE implements transparent, end-to-end encryption, so there is no impact for InputFormat.

3.6 Upgrade

If customer upgrade to hawq new version and does not enable TDE, there is no upgrade.

If customer upgrade to hawq new version and want to enable TDE, it need to initialize new hawq cluster and then migrate existing data to new cluster.

4 Test Plan

Test Case	**Description**
AES test	
Communicate with KMS	
Read/Write file	Write first, and read it, judge equal.
Append file many times	
Copy file	
Move file	Ok in the same encryption zone
Big file (> 64M)	Multi blocks
Authenticate Plain/Simple 1. Kerberos 2. Token	
Kms acl	
Performance Test	Compare results of performance test with implement TDE before

More test case can refer to **hdfs tde test plan**, see reference 3.

5 Reference

1. A good ppt of introduce to TDE
   https://drive.google.com/open?id=0B4RCw2E2M2ZPRXlrd08tVTlFQjQ
2. Hadoop TDE official introduce
   http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html
3. HDFS TDE jira (include patch, design doc and test plan)
   https://issues.apache.org/jira/browse/HDFS-6134
4. 『Professional Hadoop』
   http://www.wrox.com/WileyCDA/WroxTitle/Professional-Hadoop.productCd-111926717X.html
5. A personal TDE implement
   https://github.com/bdrosen96/libhdfs3
6. Hdfs append
   https://issues.apache.org/jira/secure/attachment/12445209/appendDesign3.pdf