Summary

The **Cross-site Big Table Solution for HBase** offers you the facilities to query, modify and scan as well as perform administrative operations of big tables which are physically stored across a group of geographically distributed HBase clusters (referred to as **HBase Cluster Group** below).
The enhanced interfaces are divided into four categories:

1. Enabling and configuration of the Cross-site Big Table Solution for HBase.
2. Query and modification of Cross-site Big Tables.
3. Administration of Cross-site Big Tables.
4. Administration of the HBase Cluster Group.

1. Enabling and configuration of Cross-site Big Table Solution for HBase:

To enable this function, set the following properties in `/etc/hbase/conf/hbase-site.xml` accordingly on all nodes of the targeted HBase clusters:

```
<property>
  <name>hbase.use.partition.table</name>
  <value>true</name>
<property>
<property>
  <name>hbase.partition.zookeeper</name>
  <value>{quorum}:{client_port}:{znode_parent}</name>
  <description>
    The address string of the ZooKeeper quorum that manages the Data Center
    Group information.
  </description>
<property>
```

Meanwhile, the following property should be set on the HBase client side:

```
<property>
  <name>hbase.use.partition.table</name>
  <value>true</name>
<property>
```

In addition, there is an optional client-side property which decides whether cross-site big table administration and cross-site big table scan operations should ignore unavailable HBase clusters instead of throwing an exception:

```
<property>
  <name>hbase.partition.ignore.unavailable.clusters</name>
  <value>{true|false}</name>
<property>
```

2. Query and modification of Cross-site Big Tables:

The physical storage location of a cross-site big table row is decided by the cluster name produced by the mapping from the row key of this row. The mapping rule is specified by an implementation class of the interface `ClusterLocator` provided at the time of cross-site big table creation (see section 3). A cluster name uniquely identifies an HBase cluster in the cluster group. See section 4 for the administration interfaces of the HBase Cluster Group. An attempt to get or put or delete a table row that does not fall into any of the existing clusters in the group would cause an IOException.
Hence, all data query and modification interfaces for cross-site big tables are exactly the same as for ordinary HTables. The routing of physical HBase data for cross-site big tables is transparent to HBase clients.

```
package org.apache.hadoop.hbase.partition;
public interface ClusterLocator extends Writable {
  public String getClusterName(final byte[] row) throws
RowNotLocatableException ;
}
```

Our HBase package now includes four built-in implementation classes:
`PrefixClusterLocator`, `SuffixClusterLocator`, `SubstringClusterLocator` and `CompositeSubstringClusterLocator`.
The `PrefixClusterLocator` returns a subsequence of characters of the row key from its start to the character before the first appearance of the delimiter (default ","), which can be set in constructor.
The `SuffixClusterLocator` returns a subsequence of the row key from the character after the last appearance of the delimiter (can be set in constructor, default ",") to the end.
The `SubstringClusterLocator` returns a substring value of the row key. Both the start and the end position of the substring can be specified in constructor.
The `CompositeSubstringClusterLocator` returns the n^{th} substring value of the row key, split by the delimiter (can be set in constructor, default",").
Examples:
`new PrefixClusterLocator(":").getClusterName("310:940c5")` returns "310".
`new SuffixClusterLocator(".").getClusterName("940c5.020")` returns "020".
`new SubstringClusterLocator(3, 6).getClusterName("940a0165")` returns "a01".
`new CompositeSubstringClusterLocator("_", 1).getClusterName("r96_c2_p")` returns "c2".

Let's look at an example of how the cluster locator mechanism works. First we create a cross-site big table "BTable01" and specifies `SuffixClusterLocator(",")` as the table's `ClusterLocator` instance by passing it to the `createTable` method. Now we are going to put a table row with row key "D52,A1234,20120101,001" into the big table "BTable01". The `SuffixClusterLocator` produces "001" from the above row key, and thus the "put" action

of this table row is automatically forwarded to cluster “001”, say, with ZooKeeper quorum address “hbase01.sh.intel.com,hbase02.sh.intel.com,hbase03.sh.intel.com”.

An extended set of interfaces in class `Scan` provides the user with a way to filter out clusters that carry irrelevant table rows. By calling these methods (listed below), a cross-site big table scan can be confined to only a specific list of clusters in the cluster group instead of all clusters in the group.

```
package org.apache.hadoop.hbase.client;
public class Scan implements Writable {
  public void addCluster(String cluster);
  public void setClusterList(List<String> clusters);
  public List<String> getClusterList();
  public boolean hasClusterList();
  public List<String> getUnavailableClusterList();
}
```

A cross-site big table scan is also influenced by the option `hbase.partition.ignore.unavailable.clusters` (see section 1).
If one or more of the clusters in the cluster group are currently unavailable to the HBase client, the scan would ignore connection errors and retrieve data from those running clusters when this option is set true. Still you can get the list of unavailable clusters by calling `scan.getUnavailableClusterList()` after the call to `htable.getScanner(scan)` returns.
An IOException would be thrown if the option is set false.

3. Administration of Cross-site Big Tables:

To create a cross-site big table, a `ClusterLocator` object should be passed on to the `createTable` method in addition to other parameters. The `ClusterLocator` object is to specify the mapping rule from a table row key to its corresponding cluster name (see section 2). A virtual table will be created, along with physical HTables on each of the clusters in the cluster group. The split keys specified at the creation of a cross-site big table will be applied to the creation of all corresponding physical HTables.
The call to `listTables()` will return a list of all cross-site big tables and ordinary HTables.
All other table administration methods for cross-site big tables have the same usage and signatures, as listed below.
Option `hbase.partition.ignore.unavailable.clusters` (see section 1) can be set true to avoid getting an IOException for unavailable clusters when performing calls to `createTable`, `deleteTable`, `enableTable`, `disableTable`, `isTableEnabled` or `isTableDisabled`. The creation, deletion, enabling or disabling of a cross-site big table will take effect immediately on an unavailable cluster once the cluster is brought up to service again.

```
package org.apache.hadoop.hbase.client;
public class HBaseAdmin implements Abortable {
  public HTableDescriptor[] listTables() throws IOException;
  public void createTable(HTableDescriptor desc, ClusterLocator locator)
throws IOException;
  public void createTable(HTableDescriptor desc, byte [] startKey, byte
[] endKey, int numRegions, ClusterLocator locator) throws IOException;
  public void createTable(HTableDescriptor desc, byte [][] splitKeys,
ClusterLocator locator) throws IOException;
  public void createTableAsync(HTableDescriptor desc, byte [][] splitKeys,
ClusterLocator locator) throws IOException;
  public void deleteTable(final String tableName) throws IOException;
  public void deleteTable(final byte [] tableName) throws IOException;
  public void enableTable(final String tableName) throws IOException;
  public void enableTable(final byte [] tableName) throws IOException;
  public void enableTableAsync(final String tableName) throws
IOException;
  public void enableTableAsync(final byte [] tableName) throws
IOException;
  public void disableTableAsync(final String tableName) throws
IOException;
  public void disableTableAsync(final byte [] tableName) throws
IOException;
  public void disableTable(final String tableName) throws IOException;
  public void disableTable(final byte [] tableName) throws IOException;
  public boolean isTableEnabled(String tableName) throws IOException;
  public boolean isTableEnabled(byte[] tableName) throws IOException;
  public boolean isTableDisabled(final String tableName) throws
IOException;
  public boolean isTableDisabled(byte[] tableName) throws IOException;
  public boolean isTableAvailable(byte[] tableName) throws IOException;
  public boolean isTableAvailable(String tableName) throws IOException;
  public void addColumn(final String tableName, HColumnDescriptor column)
throws IOException;
  public void addColumn(final byte [] tableName, HColumnDescriptor column)
throws IOException;
  public void deleteColumn(final String tableName, final String
columnName) throws IOException;
  public void deleteColumn(final byte [] tableName, final byte []
columnName) throws IOException;
  public void modifyColumn(final String tableName, final String
columnName, HColumnDescriptor descriptor) throws IOException;
  public void modifyColumn(final String tableName, HColumnDescriptor
descriptor) throws IOException;
```

```
  public void modifyColumn(final byte [] tableName, final byte []
columnName, HColumnDescriptor descriptor) throws IOException;
  public void modifyColumn(final byte [] tableName, HColumnDescriptor
descriptor) throws IOException;
  public void modifyTable(final byte [] tableName, HTableDescriptor htd)
throws IOException;
  public void flush(final String tableNameOrRegionName) throws
IOException, InterruptedException;
  public void flush(final byte [] tableNameOrRegionName) throws
IOException, InterruptedException;
  public void compact(final String tableNameOrRegionName) throws
IOException, InterruptedException;
  public void majorCompact(final String tableNameOrRegionName) throws
IOException, InterruptedException;
  public void majorCompact(final byte [] tableNameOrRegionName) throws
IOException, InterruptedException;
  public void split(final String tableNameOrRegionName) throws
IOException, InterruptedException;
  public void split(final byte [] tableNameOrRegionName) throws
IOException, InterruptedException;
  public void split(final byte [] tableNameOrRegionName, final byte []
splitPoint) throws IOException, InterruptedException;
}
```

4. Administration of the HBase Cluster Group:

Class PartitionAdmin provides methods for adding and removing a cluster as well as getting a list of all existing clusters in the HBase Cluster Group.
When a new cluster is added to the group, all physical HTables needed by the existing cross-site big tables will be created automatically. However, when a cluster is removed from the group, no clean-up will be performed and all HTables created before will remain.
The format of a cluster address string is as follows:
{ZooKeeper_Quorum}:{ZooKeeper_Client_Port}:{HBase_ZooKeeper_Node_Parent}
Example: “hbase01.sh.intel.com,hbase02.sh.intel.com,hbase03.sh.intel.com:2181:/hbase”

```
package org.apache.hadoop.hbase.client.partition;
public class PartitionAdmin {
  public PartitionAdmin(final Configuration conf) throws IOException;
  public void addCluster(String clusterName, String clusterAddress)
throws IOException;
  public void removeCluster(String clusterName) throws IOException;
  public Map<String, String> listClusters() throws IOException;
  public ClusterStatus getClusterStatus(String clusterName) throws
IOException;
```

```
}
```

Code examples:

```
Configuration conf = HBaseConfiguration.create();
PartitionAdmin admin = new PartitionAdmin(conf);
String clusterAddr1 = "hb01,hb02,hb03:2181:/hbase";
String clusterAddr2 = "hb11,hb12,hb13:2181:/hbase";
// Add the first cluster to the cluster group.
admin.addCluster("001", clusterAddr1);
// Add the second cluster to the cluster group.
admin.addCluster("011", clusterAddr2);

HBaseAdmin hba = new HBaseAdmin(conf);
String tableName = "OurFirstBigTable";
HTableDescriptor desc = new HTableDescriptor(tableName);
desc.addFamily(new HColumnDescriptor("fam1"));
desc.addFamily(new HColumnDescriptor("fam2"));
// Create a cross-site big table "OurFirstBigTable".
hba.createTable(desc, new SubstringClusterLocator(6, 9));

HTable table = new HTable(tableName);
table.put(new Put(Bytes.toBytes("r00002011"))); // stored in the second
cluster: 011
table.put(new Put(Bytes.toBytes("r00004001"))); // stored in the first
cluster: 001
table.put(new Put(Bytes.toBytes("r00001001"))); // stored in the first
cluster: 001
table.put(new Put(Bytes.toBytes("r00001011"))); // stored in the second
cluster: 011
try {
    table.put(new Put(Bytes.toBytes("r00002021")));
} catch (ClusterNotFoundException e) {
    System.out.println(
            "Could not put this table row because cluster '"
            + e.getMessage() + "' does not exist in the cluster group.");
}

Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
Result r = scanner.next();
while (r != null) {
    System.out.println("ROW: " + Bytes.toString(r.getRow()));
    r = scanner.next();
```

```
}
List<String> unavailables = scan.getUnavailableClusterList();
if (unavailables.isEmpty()) {
    System.out.println("All clusters have worked properly.");
}
else {
    System.out.println("Unavailable clusters:");
    for (String cluster : unavailables) {
        System.out.println(cluster);
    }
}
scanner.close();
```

The expected output of the above code is:

```
Could not put this table row because cluster '021' does not exist in the
cluster group.
ROW: r00001001
ROW: r00001011
ROW: r00002011
ROW: r00004001
```