Roshan Naik
February 24, 2014

Hive Streaming API – Description and Spec

Traditionally adding new data into hive requires gathering a large amount of data onto HDFS and then periodically adding a new partition. This is essentially a "batch insertion". Insertion of new data into an existing is not permitted. Hive Streaming API allows data to be pumped continuously into Hive. The incoming data can be continuously committed in small batches (of records) into a Hive partition. Once data is committed it becomes immediately visible to all Hive queries initiated subsequently.

This API is intended for streaming clients such as Flume and Storm, which continuously generate data. Streaming support is built on top of ACID based insert/update support in Hive. These APIs will be made available as part of `hive-streaming` maven module which will be the only direct dependency for streaming clients.

The Classes and interfaces part of the Hive streaming API are broadly categorized into two. First set provides support for connection and transaction management while the second set provides I/O support.

Transaction and Connection management

Class `HiveEndPoint` describes a Hive End Point to connect to. This describes the database, table and partition names. Invoking the `newConnection` method on it establishes a connection to the Hive MetaStore for streaming purposes. It returns a `StreamingConnection` object. Multiple connections can be established on the same endpoint. `StreamingConnection` can then be used to initiate new transactions for performing I/O.

It is very likely that a setup where data is being streamed continuously, the data is added into new partitions periodically. Either the Hive admin have the necessary partitions pre-created or have the streaming clients create it as needed. `HiveEndPoing.newConnection()` accepts a boolean argument to indicate if the partition should be auto created. Partition creation being an atomic action, multiple clients can race to create the partition, but only one would to succeed, so streaming clients do not have to synchronize when creating a partition.

Transactions are implemented slightly differently than traditional database systems. Each transaction has an id and multiple transactions are grouped into a "Transaction Batch". After connection, a streaming client first requests for a new batch of transactions. In response it receives a set of Transaction Ids that are part of the transaction batch. Subsequently the client proceeds to consume one transaction id at a time by initiating new Transactions. Client will write() one or more records per transactions and either commits or aborts the current transaction before switching to the next one. Each TransactionBatch.write() invocation automatically associates the I/O attempt with the current Txn ID.

Concurrency Note: I/O can be performed on multiple TransactionBatches concurrently.

However the transactions within a transaction batch much be consumed sequentially.

```
public class HiveEndPoint {
  public final String metaStoreUri;
  public final String database;
  public final String table;
  public final List<String> partitionVals;

  public HiveEndPoint( String metaStoreUri
                     , String database, String table
                     , List<String> partitionVals);

  // Connects to the end point and returns the connection object. Second
argument can be used to auto create the partition if it does not exist
  public StreamingConnection newConnection(String user
                                        , boolean createPartIfNotExists);

  // Enable use with hash tables
  @Override
  public int hashCode();

  @Override
  public boolean equals(Object rhs);
}
```

```
// The Streaming connection – for acquiring Transaction Batches

public interface StreamingConnection {
   // Acquire a set of transactions
   public TransactionBatch fetchTransactionBatch(int numTransactionsHint
                                                 , RecordWriter writer)
           throws ConnectionError, StreamingException;

  // Close connection
   public void close();
}
```

***** fetchTransactionBatch(int numTransactions) throws ...**
Acquires a new batch of transactions from Hive.
numTransactions is a hint from client indicating how many transactions client needs

***** close()**
Close any open connections and resources.

Roshan Naik
February 24, 2014

```
// TransactionBatch is used to start/commit/abort a Txn in the batch
// and also for writing to the current Txn using the specified writer

public interface TransactionBatch  {
  public enum TxnState {INACTIVE, OPEN, COMMITTED, ABORTED }

  public void beginNextTransaction() throws StreamingException;

  public TxnState getCurrentTransactionState();
  public void commit() throws StreamingException;
  public void abort() throws StreamingException;

  public int remainingTransactions();
  public Long getCurrentTxnId();

  // Write Data for current Txn //
  public void write(byte[] record) throws ConnectionError, IOException,
StreamingException;
  public void write(Collection<byte[]> records) throws ConnectionError,
IOException, StreamingException;

  // Close batch
  public void close();
}
```

***** beginNextTransaction(...) throws ...**
Switch to the next transaction in the batch.
returns false if there are no more transactions.

***** commit()**
Commits the currently open transaction.

***** abort() throws ...**
Aborts the currently open transaction.

***** write(byte[] record)**
Write a record.

***** write(Collection<byte[]> records)**
Write multiple records

***** getTransactionState()**
Get the current state of the transaction

```
*** remainingTransactions()
Get a count of the unused transactions in the batch acquired by the last
call to fetchTransactionBatch. Current transaction is not considered part
of remaining transactions.


*** close()
Close the transacton batch
```

I/O – Writing Data

These classes and interfaces provide support for writing the data to Hive within a transaction. `RecordWriter` is the base interface implemented by all Writers. A Writer is responsible for taking a record in the form of a byte[] containing data in a known format (e.g. CSV) and writing it out in the format supported by Hive streaming. A RecordWriter may reorder or drop fields from the incoming record if necessary to map them to the corresponding columns in the Hive Table. A streaming client will instantiate an appropriate RecordWriter type and pass it to TransactionBatch. The streaming client does not directly interact with RecordWriter therafter. The TransactionBatch will thereafter use and manage the RecordWriter instance to perform I/O.

```
public interface RecordWriter {

   public void write(long transactionId, byte[] record)
                   throws StreamingException;

   public void flush() throws StreamingException;

   public void newBatch() throws StreamingException;

   public void closeBatch() throws StreamingException;
}
```

A `RecordWriter` has two primary functions.

a) Modify input record: This may involve, dropping fields from input data if they don't have corresponding table columns, adding nulls in case of missing fields for certain columns, and changing the order of incoming fields to match the order of fields in the table. This task requires understanding of incoming data format.
b) Encode modified record: The encoding often involves serialization using an appropriate Hive Serde. This task is agnostic of the incoming data format.

Class `DelimitedInputWriter` provides support for writing out input data that is in delimited formats (such as CSV). Class `DelimitedInputWriter` only implements the input format specific task of modifying input record.

```
public class DelimitedInputWriter extends AbstractLazySimpleRecordWriter {

  public DelimitedInputWriter(List<String> colNamesForFields
                 , String delimiter, HiveEndPoint endPoint) throws ...;

  protected byte[] reorderFields(byte[] record) throws ...;
}
```

The input format agnostic task of encoding the modified record is delegated to its abstract base class `AbstractLazySimpleRecordWriter`. This class provides bulk of the RecordWriter implementation.

```
public abstract class AbstractLazySimpleRecordWriter implements
RecordWriter {

  public final static char outputFieldSeparator = ...;


  protected AbstractLazySimpleRecordWriter(HiveEndPoint endPoint)
  throws ... ;

  protected abstract byte[] reorderFields(byte[] record)
  throws ...;

  public ArrayList<String> getTableColumns();

  @Override
  public void write(long transactionId, int bucket, byte[] record)
  throws ...;

  @Override
  public void flush() throws ...;


  @Override
  public void newBatch() throws ...;

  @Override
  public void closeBatch() throws ... ;

}
```

Example

```
///// Stream five records in two transactions /////
```

Roshan Naik
February 24, 2014

```
// Assumed HIVE table Schema:
// partitions(continent: string, country: string)
// columns(id: int, msg: string)

String dbName = "testing";
String tblName = "alerts";
ArrayList<String> partitionVals = new ArrayList<String>(2);
partitionVals.add("Asia");
partitionVals.add("India");


HiveEndPoint hiveEP = new HiveEndPoint("thrift://x.y.com:9083"
                                       , dbName, tblName, partitionVals);
StreamingConnection connection = hiveEP.newConnection(user, true);


DelimitedInputWriter writer =
                      new DelimitedInputWriter(fieldNames,",", endPt);
TransactionBatch txnBatch = connection.fetchTransactionBatch(10, writer);

///// First TXN
txnBatch.beginNextTransaction();
txnBatch.write("1,Hello streaming".getBytes());
txnBatch.write("2,Welcome to streaming".getBytes());
txnBatch.commit();

///// Second TXN
txnBatch.beginNextTransaction();
txnBatch.write("3,Roshan Naik".getBytes());
txnBatch.write("4,Alan Gates".getBytes());
txnBatch.write("5,Owen O'Malley".getBytes());
txnBatch.commit();

txnBatch.close();
connection.close();
```