Roshan Naik
Jun 6th 2014

Hive Streaming API – high level description

Relates to: HIVE-5687

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.

The classes and interfaces part of the Hive streaming API are broadly categorized into two. The first set provides support for connection and transaction management while the second set provides I/O support. Transactions are managed by the Hive MetaStore. Writes are performed to HDFS via Hive wrapper APIs that bypass MetaStore.

Note on packaging: The APIs are defined in the Java package `org.apache.hive.hcatalog.streaming` and part of the *hive-streaming* maven module in hive.

***STREAMING REQUIREMENTS*:**
A few things are currently required to use streaming.

1) Currently, only ORC storage format is supported. So "`stored as orc`" must be specified during table creation.
2) The hive table must be bucketed, but not sorted. So something like "`clustered by (colName) into 10 buckets`" must be specified during table creation. The number of buckets is ideally the same as the number of concurrent streaming writers.
3) User of the client streaming process must have the necessary permissions to write to the table or partition and create partitions in the table.
4) (Temporary requirement) When issuing queries on streaming tables, the client needs to set
   a) `hive.input.format` to `org.apache.hadoop.hive.ql.io.HiveInputFormat`
   b) `hive.vectorized.execution.enabled` to `false`
5) Settings required in hive-site.xml for MetaStore
   a) hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
   b) hive.support.concurrency = true
   c) hive.compactor.initiator.on = true
   d) hive.compactor.worker.threads > 0

Note: Streaming to unpartitioned tables is also supported.

Limitations:
- Currently, out of the box, the streaming API only provides two implementations of the

RecordWriter interface. One to handle delimited input data (such as CSV, tab separated etc) and the other for JSON (strict syntax). Support for other input formats can be provided by additional implementations of the RecordWriter interface.

1. Transaction and Connection management

The class `HiveEndPoint` describes a Hive End Point to connect to. An endpoint is either a Hive table or partition. And endpoint is cheap to create and does not internally hold on to any network connections. Invoking the `newConnection` method on it creates a new 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.

Dynamic Partition Creation: It is very likely that a setup in which data is being streamed continuously (e.g. Flume), it is desirable to have new partitions created automatically (say on a hourly basis). In such cases requiring the Hive admin to pre-create the necessary partitions may not be reasonable. Consequently the streaming API allows streaming clients to create partitions as needed. `HiveEndPoind.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 succeed, so streaming clients need not synchronize when creating a partition. The user of the client process needs to be given write permissions on the Hive table in order to create partitions.

Batching Transactions: Transactions are implemented slightly differently than traditional database systems. Multiple transactions are grouped into a "Transaction Batch" and each transaction has an id. Data from each transaction batch gets a single file on Hdfs, which eventually gets compacted with other files into a larger file automatically for efficiency.

Basic Steps: After connection is established, 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 at a time by initiating new Transactions. Client will write() one or more records per transactions and either commit or abort the current transaction before switching to the next one. Each TransactionBatch.write() invocation automatically associates the I/O attempt with the current Txn ID. The user of the streaming client (or the proxy user) needs to have write permissions to the partition or table.

Concurrency Note: I/O can be performed on multiple TransactionBatches concurrently. However the transactions within a transaction batch much be consumed sequentially.

```
package org.apache.hive.streaming;

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 end point and returns a connection object. If argument is
true, the partition indicated in the endpoint will be created
  public StreamingConnection newConnection(
                       boolean createPartIfNotExists) throws ...;

  // Enables use with hashed tables & sets
  @Override
  public int hashCode();

  @Override
  public boolean equals(Object rhs);

  // Prints readable representation of the end point
  @Override
  public String toString();

}
```

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

Arguments to constructor specify the Hive table or partition to which client intends to stream. For tables without partitions, partitionVals can be set to null or an empty list.

*** **newConnection**(boolean createPartIfNotExists) throws ...

Returns a connection object.

```
// 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,
                                    RecordWriter writer)** throws ...
Acquires a new batch of transactions from Hive.
numTransactions is a hint from client indicating how many transactions

client needs. More than one TransactionBatch cannot require separate writer instances. The writer instance can be shared with another TransactionBatch, to the same endpoint, only after the first TransactionBatch has been closed.

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

```
// 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();

  // Prints readable representation of the end point
  // public String toString();

}
```

***** 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)
Writes the 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. Current transaction
is not considered part of remaining transactions.


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

2. 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 `StreamingConnection.fetchTransactionBatch()`. The streaming client does not directly interact with `RecordWriter` therafter. The TransactionBatch will use and manage the `RecordWriter` instance to perform I/O thereafter.

```
public interface RecordWriter {
   // Invoked by TransactionBatch.write()
   public void write(long transactionId, byte[] record)
                   throws StreamingException;

   // Flush buffered writes. Invoked by TransactionBatch.commit()
   public void flush() throws StreamingException;

   // Clear buffered writes. Invoked by TransactionBatch.abort()
   public void clear() throws StreamingException;

   // Invoked via StreamingConnection.fetchTransactionBatch()
   public void newBatch() throws StreamingException;

   // Invoked by TransactionBatch.close()
   public void closeBatch() throws StreamingException;
```

```
}
```

A `RecordWriter`'s write() method has three primary functions.

a) **Modify**: Optionally 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 typically requires understanding of incoming data format and table schema awareness.
b) **Encode**: Next the record needs to be encoded in a manner that is understood by the streaming subsystem. This requires the use of a Hive Serde (best suited to handle the input data format) and is not dependent on the actual serde specified on the Hive table.
c) **Write**: Write the encoded record to the endpoint.

Class `DelimitedInputWriter` implements `RecordWriter` interface to provide support for simple delimited input data formats (such as CSV, tab separated etc). Delimiter is customizable. It internally uses LazySimpleSerde.

```
public class DelimitedInputWriter
                          extends AbstractRecordWriter {

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

  public DelimitedInputWriter(List<String> colNamesForFields
                 , String delimiter, HiveEndPoint endpoint
                 , char serdeSeparator) throws ...;

  @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 ... ;
}
```

Class `StrictJsonWriter` implements `RecordWriter` interface to provide support for streaming JSON (strict syntax, UTF8) records. Delimiter is customizable. It internally uses org.apache.hive.hcatalog.data.JsonSerDe. The object names specified in the JSON record are directly mapped to the corresponding columns in the table.

```
public class StrictJsonWriter
                        extends AbstractRecordWriter {

  public StrictJsonWriter (HiveEndPoint endPoint) throws ...;


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

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

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

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

Example

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

// Assumed HIVE table Schema:
create table alerts ( id int , msg string )
     partitioned by (continent string, country string)
     clustered by (id) into 5 buckets
     stored as orc; // currently ORC is required for streaming


//-------   MAIN THREAD  ------- //
String dbName = "testing";
String tblName = "alerts";
ArrayList<String> partitionVals = new ArrayList<String>(2);
partitionVals.add("Asia");
partitionVals.add("India");
String serdeClass = "org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe";


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

.. spin up threads..


//-------   Thread 1  -------//
StreamingConnection connection = hiveEP.newConnection(true);
DelimitedInputWriter writer =
```

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

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

if(txnBatch.remainingTransactions() > 0) {
  ///// Batch 1 - 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();
}

txnBatch = connection.fetchTransactionBatch(10, writer);

///// Batch 2 - First TXN
txnBatch.beginNextTransaction();
txnBatch.write("6,David Schorow".getBytes());
txnBatch.write("7,Sushant Sowmyan".getBytes());
txnBatch.commit();

if(txnBatch.remainingTransactions() > 0) {
  ///// Batch 2 - Second TXN
  txnBatch.beginNextTransaction();
  txnBatch.write("8,Ashutosh Chauhan".getBytes());
  txnBatch.write("9,Thejas Nair" getBytes());
  txnBatch.commit();

  txnBatch.close();
}

connection.close();




//-------   Thread 2  -------//

StreamingConnection connection2 = hiveEP.newConnection(true);
DelimitedInputWriter writer2 =
                    new DelimitedInputWriter(fieldNames, ",", endPt);
TransactionBatch txnBatch2= connection.fetchTransactionBatch(10, writer2);

///// Batch 1 - First TXN
txnBatch2.beginNextTransaction();
```

```
txnBatch2.write("21,Venkat Ranganathan".getBytes());
txnBatch2.write("22,Bowen Zhang".getBytes());
txnBatch2.commit();

///// Batch 1 - Second TXN
if(txnBatch2.remainingTransactions() > 0) {
  txnBatch2.beginNextTransaction();
  txnBatch2.write("23,Venkatesh Seetaram".getBytes());
  txnBatch2.write("24,Deepesh Khandelwal".getBytes());
  txnBatch2.commit();
}

txnBatch2.close();
connection.close();


txnBatch = connection.fetchTransactionBatch(10, writer);

///// Batch 2 - First TXN
txnBatch.beginNextTransaction();
txnBatch.write("26,David Schorow".getBytes());
txnBatch.write("27,Sushant Sowmyan".getBytes());
txnBatch.commit();


txnBatch2.close();

connection2.close();
```