Streaming Data Ingest V2

Starting in release Hive 3.0.0, Streaming Data Ingest is deprecated and is replaced by newer V2 API (HIVE-19205).

Hive Streaming API

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”.

Hive Streaming API allows data to be pumped continuously into Hive. The incoming data can be continuously committed in small batches of records into an existing Hive partition or table. Once data is committed it becomes immediately visible to all Hive queries initiated subsequently.

This API is intended for streaming clients such as NiFi, Flume and Storm, which continuously generate data. Streaming support is built on top of ACID based insert/update support in Hive (see Hive Transactions).

The Classes and interfaces part of the Hive streaming API are broadly categorized into two sets. The first set provides support for connection and transaction management while the second set provides I/O support. Transactions are managed by the metastore. Writes are performed directly to destination filesystem defined by the table (HDFS, S3A etc.).

Streaming to unpartitioned tables, partitioned table with static partitions and partitioned table with dynamic partitions are all supported. The API supports Kerberos authentication and Storage based authorization. The client user has to be logged in using kerberos before invoking the API. The logged in user should have appropriate storage permissions to write to destination partition or table location. It is recommended to use 'hive' user in order for the hive queries to be able to read the data back (written by streaming API) with doAs set to false (query is run as hive user).

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

Streaming Mutation API Deprecation and Removal

Starting in release 3.0.0, Hive deprecated Streaming Mutation API from hive-hcatalog-streaming module and will no longer be supported in future releases. The new hive-streaming module no longer support mutation API.

Streaming Requirements

A few things are required to use streaming.

1. The following settings are required in hive-site.xml to enable ACID support for streaming:
    a. **hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager**
    b. **hive.compactor.initiator.on = true** (See more important details here)
    c. **hive.compactor.worker.threads** > **0**
2. *“stored as orc”* must be specified during table creation. Only ORC storage format is supported currently.
3. tblproperties("transactional"="true") must be set on the table during creation.
4. User of the client streaming process must have the necessary permissions to write to the table or partition and create partitions in the table.

Limitations

Out of the box, currently, the streaming API only provides support for streaming delimited input data (such as CSV, tab separated, etc.), JSON and Regex formatted data. The records writers that are supported in Hive 3.0.0 release are

```
StrictDelimitedInputWriter

StrictJsonWriter

StrictRegexWriter
```

All these record writers expects strict schema match, meaning the schema of the records should exactly match with the table schema (Note: The writers does not perform schema check, it is up to the clients to make sure the record schema matches the table schema).

Support for other input formats can be provided by additional implementations of the *RecordWriter* interface.

Currently only ORC is supported for the format of the destination table.

API Usage

Transaction and Connection Management

HiveStreamingConnection

The class HiveStreamingConnection describes the streaming connection related information. This describes the database, table, partition names, metastore URI to connect to and record writer to use for streaming records into destination partition or table. All these information can be specified via Builder API which then establishes a connection to the Hive MetaStore for streaming purposes. Invoking connect on the Builder API returns a StreamingConnection object. StreamingConnection can then be used to initiate new transactions for performing I/O.

```
* CONCURRENCY NOTE: The streaming connection APIs and record writer APIs are not thread-safe. Streaming
connection creation,
* begin/commit/abort transactions, write and close has to be called in the same thread. If close() or
* abortTransaction() has to be triggered from a separate thread it has to be co-ordinated via external variables
or
* synchronization mechanism
```

HiveStreamingConnection API also supports 2 partitioning mode (static vs dynamic). In static partitioning mode, partition column values can be specified upfront via builder API. If the static partition exists (pre-created by the user or already existing partition in a table), the streaming connection will use it and if it does not exist the streaming connection will create a new static partition using the specified values. If the table is partitioned and if static partition values are not specified in builder API, hive streaming connection will use dynamic partitioning mode under which hive streaming connection expects the partition values to be the last columns in the record (similar to how hive dynamic partitioning works). For example, if table is partitioned by 2 columns (year and month), hive streaming connection will extract last 2 columns from the input records using which partitions will be created dynamically in metastore.

Transactions are implemented slightly differently than traditional database systems. Each transaction has an id and multiple transactions are grouped into a “Transaction Batch”. This helps grouping records from multiple transactions into fewer files (rather than 1 file per transaction). During hive streaming connection creation, transaction batch size can be specified via builder API. Transaction management is completely hidden behind the API, in most cases users do not have to worry about tuning the transaction batch size (which is an expert level setting and might not be honored in future release). Also the API automatically rolls over to next transaction batch on beginTransaction() invocation if the current transaction batch is exhausted. The recommendation is to leave the transaction batch size at default value of 1 and group several thousands records together under a each transaction. Since each transaction corresponds to a delta directory in the filesystem, committing transaction too often can end up creating too many small directories.

Transactions in a TransactionBatch are eventually expired by the Metastore if not committed or aborted after hive.txn.timeout secs. In order to keep the transactions alive, HiveStreamingConnection has a heartbeater thread which by default sends heartbeat after (hive.txn.timeout/2) intervals for all the open transactions.

See the Javadoc for HiveStreamingConnection for more information.

Usage Guidelines

Generally, the more records are included in each transaction the more throughput can be achieved. It's common to commit either after a certain number of records or after a certain time interval, whichever comes first. The later ensures that when event flow rate is variable, transactions don't stay open too long. There is no practical limit on how much data can be included in a single transaction. The only concern is amount of data which will need to be replayed if the transaction fails. The concept of a TransactionBatch serves to reduce the number of files (and delta directories) created by HiveStreamingConnection API in the filesystem. Since all transactions in a given transaction batch write to the same physical file (per bucket), a partition can only be compacted up to the the level of the earliest transaction of any batch which contains an open transaction. Thus TransactionBatches should not be made excessively large. It makes sense to include a timer to close a TransactionBatch (even if it has unused transactions) after some amount of time.

The HiveStreamingConnection is highly optimized for write throughput (Delta Streaming Optimizations) and as a result the delta files generated by Hive streaming ingest have many of the ORC features disabled (dictionary encoding, indexes, compression, etc.) to facilitate high throughput writes. When the compactor kicks in, these delta files get rewritten into read- and storage-optimized ORC format (enable dictionary encoding, indexes and compression). So it is recommended to configure the compactor more aggressively/frequently (refer to Compactor) to generate compacted and optimized ORC files.

Notes about the HiveConf Object

HiveStreamingConnect builder API accepts a HiveConf argument. This can either be set to null, or a pre-created HiveConf object can be provided. If this is null, a HiveConf object will be created internally and used for the connection. When a HiveConf object is instantiated, if the directory containing the hive-site.xml is part of the java classpath, then the HiveConf object will be initialized with values from it. If no hive-site.xml is found, then the object will be initialized with defaults. Pre-creating this object and reusing it across multiple connections may have a noticeable impact on performance if connections are being opened very frequently (for example several times a second). Secure connection relies on 'metastore.kerberos.principal' being set correctly in the HiveConf object.

Regardless of what values are set in hive-site.xml or custom HiveConf, the API will internally override some settings in it to ensure correct streaming behavior. The below is the list of settings that are overridden:

- **hive.txn.manager** = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager
- **hive.support.concurrency** = true
- **hive.metastore.execute.setugi** = true
- **hive.exec.dynamic.partition.mode** = nonstrict
- **hive.exec.orc.delta.streaming.optimizations.enabled** = true
- **hive.metastore.client.cache.enabled** = false

I/O – Writing Data

These classes and interfaces provide support for writing the data to Hive within a transaction.

RecordWriter

RecordWriter is the base interface implemented by all Writers. A Writer is responsible for taking a record in the form of a byte[] (or InputStream with configurable line delimiter) containing data in a known format (such as CSV) and writing it out in the format supported by Hive streaming. A RecordWriter with Strict implementation expects record schema to exactly match as that of table schema. A RecordWriter writing in dynamic partitioning mode expects the partition columns to be the last columns in each record. If partition column value is empty or null, records will go into __HIVE_DEFAULT_PARTITION__. A streaming client will instantiate an appropriate RecordWriter type and pass it to HiveStreamingConnection builder API. The streaming client does not directly interact with RecordWriter thereafter. The StreamingConnection object will thereafter use and manage the RecordWriter instance to perform I/O. See the Javadoc for details.

A RecordWriter's primary functions are:

1. 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 adding __HIVE_DEFAULT_PARTITION__ if partition column value is null or empty. Dynamically creating partitions requires understanding of incoming data format to extract last columns to extract partition values.
2. Encode modified record: The encoding involves serialization using an appropriate Hive SerDe.
3. For bucketed tables, extract bucket column values from the record to identify the bucket where the record belongs.
4. For partitioned tables, in dynamic partitioning mode, extract the partition column values from last N columns (where N is number of partitions) of the record to identify the partition where the record belongs.
5. Write encoded record to Hive using the AcidOutputFormat's record updater for the appropriate bucket.

StrictDelimitedInputWriter

Class StrictDelimitedInputWriter implements the RecordWriter interface. It accepts input records that in delimited formats (such as CSV) and writes them to Hive. It expects the record schema to match the table schema and expects partition values at the last. The input records are converted into an Object using LazySimpleSerde to extract bucket and partition columns, which is then passed on to the underlying AcidOutputFormat's record updater for the appropriate bucket. See Javadoc.

StrictJsonWriter

Class StrictJsonWriter implements the RecordWriter interface. It accepts input records that in strict JSON format and writes them to Hive. It converts the JSON record directly into an Object using JsonSerde, which is then passed on to the underlying AcidOutputFormat's record updater for the appropriate bucket and partition. See Javadoc.

StrictRegexWriter

Class StrictRegexWriter implements the RecordWriter interface. It accepts input records, regex that in text format and writes them to Hive. It converts the text record using proper regex directly into an Object using RegexSerDe, which is then passed on to the underlying AcidOutputFormat's record updater for the appropriate bucket. See Javadoc.

AbstractRecordWriter

This is a base class that contains some of the common code needed by RecordWriter objects such as schema lookup and computing the bucket and partition into which a record should belong.

Error Handling

It's imperative for proper functioning of the system that the client of this API handle errors correctly. The API just returns StreamingException, however there are several subclasses of StreamingException that are thrown under different situations.

ConnectionError - When a connection to metastore cannot be established or when HiveStreamingConnection connect API is used incorrectly

InvalidTable - Thrown when destination table does not exists or is not ACID transactional table

InvalidTransactionState - Thrown when the transaction batch gets to invalid state

SerializationError - Thrown when SerDe throws any exception during serialization/deserialization/writing of records

StreamingIOFailure - Thrown if destination partition cannot be created, if record updaters throws any IO errors during writing or flushing of records.

TransactionError - Throw when internal transaction cannot be committed or aborted.

Its up to the clients to decide which exceptions can be retried (typically with some backoff), ignored or rethrown. Typically connection related exceptions can be retried with exponential backoff. Serialization related errors can either be thrown or ignored (if some incoming records are incorrect/corrupt and can be dropped).

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 tblproperties("transactional"="true"); // currently ORC is required for streaming


//-------   MAIN THREAD  ------- //
String dbName = "testing";
String tblName = "alerts";

.. spin up thread 1 ..
// static partition values
ArrayList<String> partitionVals = new ArrayList<String>(2);
partitionVals.add("Asia");
partitionVals.add("India");

// create delimited record writer whose schema exactly matches table schema
StrictDelimitedInputWriter writer = StrictDelimitedInputWriter.newBuilder()
                                      .withFieldDelimiter(',')
                                      .build();
// create and open streaming connection (default.src table has to exist already)
StreamingConnection connection = HiveStreamingConnection.newBuilder()
                                  .withDatabase(dbName)
                                  .withTable(tblName)
                                  .withStaticPartitionValues(partitionVals)
                                  .withAgentInfo("example-agent-1")
                                  .withRecordWriter(writer)
                                  .withHiveConf(hiveConf)
                                  .connect();
// begin a transaction, write records and commit 1st transaction
connection.beginTransaction();
connection.write("1,val1".getBytes());
connection.write("2,val2".getBytes());
connection.commitTransaction();
// begin another transaction, write more records and commit 2nd transaction
connection.beginTransaction();
connection.write("3,val3".getBytes());
connection.write("4,val4".getBytes());
connection.commitTransaction();
// close the streaming connection
connection.close();



.. spin up thread 2 ..
// dynamic partitioning
// create delimited record writer whose schema exactly matches table schema
StrictDelimitedInputWriter writer = StrictDelimitedInputWriter.newBuilder()
                                      .withFieldDelimiter(',')
                                      .build();
// create and open streaming connection (default.src table has to exist already)
StreamingConnection connection = HiveStreamingConnection.newBuilder()
                                  .withDatabase(dbName)
                                  .withTable(tblName)
                                  .withAgentInfo("example-agent-1")
                                  .withRecordWriter(writer)
                                  .withHiveConf(hiveConf)
                                  .connect();
// begin a transaction, write records and commit 1st transaction
```

```
connection.beginTransaction();
// dynamic partition mode where last 2 columns are partition values
connection.write("11,val11,Asia,China".getBytes());
connection.write("12,val12,Asia,India".getBytes());
connection.commitTransaction();
// begin another transaction, write more records and commit 2nd transaction
connection.beginTransaction();
connection.write("13,val13,Europe,Germany".getBytes());
connection.write("14,val14,Asia,India".getBytes());
connection.commitTransaction();
// close the streaming connection
connection.close();
```