HIVE-13680: Provide a way to compress ResultSets

Kevin Liew

May 24, 2016

Abstract

Hive data pipelines invariably involve JDBC/ODBC drivers which access Hive through its Thrift interface. Consequently, any enhancement to the Thrift vector will benefit all users.

Prior to HIVE-12049, HiveServer2 would read a full ResultSet from HDFS before deserializing and re-serializing into Thrift objects for RPC transfer. Following our enhancement, task nodes serialize Thrift objects from their own block of a ResultSet file. HiveServer2 reads the Thrift output and transfers it to the remote client. This parallel serialization strategy reduced latency in the data pipeline.

However, network capacity is often the most scarce resource in a system. As a further enhancement, network load can be eased by having task nodes compress their own block of a ResultSet as part of the serialization process.

1 Introduction

The changes proposed herein draw from Rohit Dholakia's design document and patches for HIVE-10438, which implemented compression on HiveServer2. Now that HIVE-12049 has been committed, compression can take place in parallel on the task nodes.

Our goals for this enhancement are to:

- improve performance out-of-the-box for new clients
- maintain compatibility with old clients
- provide flexibility yet security
- confer a simple interface

2 Design Overview

2.1 Compressor-Decompressor Interface

We define a compressor-decompressor (CompDe) interface which must be implemented by plugins. A default CompDe will process all data-types using the Snappy algorithm. Type-specific compression-decompression can be achieved by implementing a CompDe that delegates to other CompDes based on a case-switch block. CompDes may also pass specific data-types through unprocessed.

org.apache.hive.service.cli.CompDe; CompDe.java

```java
@InterfaceAudience.Private
@InterfaceStability.Stable
public interface ColumnCompDe {
  public byte[] compress(ColumnBuffer column);
  public Object decompress(TEnColumn compressedColumn);
}
```

Operating in the final task nodes, ‘compress‘ takes a batch of rows contained in a ColumnBuffer (number of rows will be equal to hive.server2.thrift.resultset.max.fetch.size except for the last batch) and outputs a binary blob. The compressor is free to pack additional details such as look-up tables within this blob.

The client receives results batch-by-batch, calling ‘decompress‘ on each compressed column to receive an Object containing the rows in that column-batch which, along with ‘nulls‘ and the type-information in TEnColumn, will be used to construct a ColumnBuffer.

2.2 Configuration options

Option	Default	Description
hive.resultSet.compressors	snappy	A list of compressors that the server can use, ordered by preference.
hive.server2.thrift.resultset.max.fetch.size	1000	Max number of rows sent in one Fetch RPC call by the server to the client.

Table 1: Server configuration options

Parameter	Default	Description
compressors	snappy	A list of compressors that the client can use.

Table 2: Client connection parameters

2.3 Client-Server Negotiation

Upon connection, the client will send a list of available compressors. The server will reply with a list of compressors ordered by preference. Both the client and server will choose the first compressor in the server's list that also exists in the client's list. All results for that session will be compressed using that compressor. The results will not be compressed if the client and server cannot agree on a compressor, or if either the client or server has configured an empty compressor list. This negotiation scheme will maintain compatibility with old clients while using compression whenever it is available.

3 Implementation

3.1 Thrift Structures

TCLIService.thrift

```
//Represents an encoded column
struct TEnColumn {
  1: required binary enData
  2: required binary nulls
  3: required TTypeId type
}

// Represents a rowset
struct TRowSet {
  // The starting row offset of this rowset.
  1: required i64 startRowOffset
  2: required list<TRow> rows
  3: optional list<TColumn> columns
  4: optional binary binaryColumns
  5: optional i32 columnCount
  6: optional binary compressorMask
}
```

TRowSet will be amended to store the compressor-name, a bitmask to track compressed columns, and a list of TEnColumn (compressed columns). Otherwise, the Thrift structures used to support compression are largely unchanged from HIVE-10438 . They are described here for completeness.

TEnColumns ‘enData‘ is a binary blob containing one compressed row-batch of a column. ‘nulls‘ is a bitmap indicating null rows in the column. ‘type‘ is the column's data-type.

TRowSet ‘startRowOffet‘ and ‘rows‘ are deprecated following HIVE-3746. Result files are now column-oriented and either in ‘binaryColumns‘ (in the task nodes and on HiveServer2) or a combination of ‘columns‘ and ‘enColumns‘ (in the client after deserializing ‘binaryColumns‘). ‘compressorMask‘ is a bit-mask indicating compressed columns. During the negotiation stage, both the server and client will use the same algorithm to choose the same CompDe, so it is not necessary to send the compressor name with TRowSet.

3.2 Compression

The output of a final node in a DAG is either a set of rows (from a map task) or a single value (from a reduce task). Following HIVE-12049 , the output is buffered row-by-row into TColumns in TRowSet and serialized in batches to a file in HDFS. The final output file is read by HiveServer2 and sent to the client.

Compressors will operate on the batch-level in ThriftJDBCBinarySerDe. In the event that a column is not compressible, the column will be serialized as an uncompressed TColumn. Compressed TEnColumn and uncompressed TColumn will be serialized contiguously in the output file. ‘compressorMask‘ will indicate which columns were compressed successfully. This allows for a robust system where each column will have a compression state per-batch. If any column fails to compress in one batch, that column can still be compressed in other batches.

3.3 Decompression

Results are serialized in batches. Consequently, the client must deserialize batch-by-batch in ColumnBasedSet. For each batch: the client starts with a check binary with a bit value for ‘1‘ and does a bitwise ‘AND‘ with ‘compressorMask‘. If the result is equal to the check bit, then the current column is compressed. The client reads either a compressed or uncompressed column into the appropriate Thrift object which is converted into ColumnBuffer. This is done for each column, with the check bit shifted to the left to check the compression state of that column for that iteration. The end result is a ColumnBasedSet for each batch.