HIVE-13680: Provide a way to compress ResultSets

Kevin Liew

May 18, 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 Interface

We will define a compressor interface which must be implemented by compressor plugins. A Snappy compressor will compress all data-types using the Snappy algorithm. Type-specific compression can be achieved by implementing a compressor that delegates compression to other compressors based on a case-switch block.

2.2 Client-Server Negotiation

Upon connection, the client will specify a list of compressors ordered by preference in the connection string's 'compressor' parameter. The server will pick the first compressor that it can find in the classpath. All results for that session will be compressed using that plugin and each result set will have a field indicating the chosen compressor. The results will not be compressed if none of the requested compressors are available to the server.

To maintain compatibility with old clients, the server will not compress results unless the client has requested compression. However, new versions of beeline will try to negotiate a compressor scheme by default.

2.3 Configuration options

Option	**Default**	**Description**
hive.resultSet .compressor .enabled	true	Enable or disable compressor negotiation.
hive.server2 .thrift.resultset .max.fetch.size	1000	Max number of rows sent in one Fetch RPC call by the server to the client. Also max number of rows in one compressed batch.

Table 1: Configuration options

3 Implementation

3.1 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 (deserializing the columns as a binary blob) and sent to the client.

Compressors will operate on the batch-level in ThriftJDBCBinarySerDe and compressed batches will be serialized contiguously in the output file.

In the event that a column is not compressible, the column will be serialized as an uncompressed column.

3.2 Decompression

Results are serialized in contiguous compressed batches. Consequently, the client must deserialize batch-by-batch.

TBD

3.3 Compressed RowSet Structures

3.3.1 HiveServer2 Structures

TBD

3.3.2 Thrift Objects

TEnColumn must now store a list of batch offsets to delimit the compressed blob and allow decompression of contiguous batches of rows within each column. TRowSet will have a list of TEnColumn to store compressed columns. Otherwise, the Thrift structures used to support compression are largely unchanged from HIVE-10438 . They are described here for completeness.

TCLIService.thrift

```
//Represents an encoded column
struct TEnColumn {
  1: required binary enData
  2: required binary nulls
  3: required TTypeId type
  4: required list<i32> batchSize
}

// 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 list<TEnColumn> enColumns
  7: optional string compressorName
}
```

TEnColumns ‘enData‘ is a binary blob containing the contiguous compressed batches within the node’s block of the output file. ‘nulls‘ is a bitmap indicating null rows in the column set. ‘type‘ is the column’s data-type. ‘batchSize‘ contains the batch size for each compressed batch.

TRowSet ‘startRowOffet‘ and ‘rows‘ are deprecated following HIVE-3746. Result files are now column-oriented and either in ‘binaryColumns‘ (on HiveServer2) or a combination of ‘columns‘ and ‘enColumns‘ (in the task nodes and in the client). ‘compressorName‘ indicates the compressor plugin that was used to compress the result set.

3.4 Compressor-Decompressor Interface

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

```
@InterfaceAudience.Private
@InterfaceStability.Stable
public interface ColumnCompDe {
  public byte[] compress(ColumnBuffer columns);
  public ColumnBuffer decompress(
    byte[] columnsBlob, int startByte, int endByte);
  public boolean isCompressible(ColumnBuffer columns);
}
```

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

TBD

4 Custom Compressors

A default compressor will be provided for Snappy compression. To use other compression algorithms or to have type-specific compression, the user must implement the CompDe interface.

TBD