Introduction

This document is a proposal for a streaming ResultSet compression protocol in HiveServer2. It outlines a set of changes to improve performance for all Thrift clients of Hive. The core idea is to enhance the transfer of data between HiveServer2 and a client by enabling implementation of a wide variety of compression techniques to compress the data transferred over Thrift. This approach benefits the broadest set of users--users accessing Hive's Thrift interface directly and users using ODBC/JDBC (which in turn use Hive's Thrift interface).

Motivation

Frameworks such as Apache Hive, Apache Spark, and other similar ones are geared towards making it possible to query very large datasets. But, after a query has been run and the ResultSet has to be sent back to the client, what happens? In case of Thrift's TBinaryProtocol, the ResultSet is sent as-is; with TCompactProtocol, there is some compression. Having said that, there is not an external way of writing compression techniques that can work with Apache Hive directly. Prudent application of ResultSet compression techniques at this boundary between client and server will accelerate the transfer of the ResultSet. In some cases, a compression library can reduce up to 50% what was being transmitted earlier.

We refer the interested reader to Carl Steinbach's design document for specifics on the shortcoming of HiveServer. Carl led the development of HiveServer2 in Hive 0.11 HIVE-2935. A follow-on enhancement by Navis Ryu changing the serialization from a row-oriented to a column-oriented format HIVE-3746 came in Hive 0.13.

While HiveServer2 is a bonafide long-lived query service, neither JIRA addressed the basic mechanism of how data moved from the server to the client. Data transmission from HiveServer2 to a client uses the Thrift serialization protocols, as mentioned above. We see both the Apache community's ODBC driver and Simba's commercial ODBC driver use the same Thrift interface. Over the past two years' development of Simba's ODBC driver, the Hive driver team at Simba has determined that driver performance is capped by this basic mechanism.

Proposed changes

Thrift API

In Hive 0.13 and earlier, a column was sent as part of a RowSet. In case of compressed columns, we want to send serialized data. Starting from protocol V6, Hive sends a rowset with columnar data. The Thrift struct looks like the following:

```
struct TRowSet {
    1: required i64 startRowOffset
    2: required list<TRow> rows
```

```
    3: optional list<TColumn> columns
}
```

As we want to send compressed columns, we need to add a list of encoded/compressed columns. We propose the following:

```
struct TRowSet {
    1: required i64 startRowOffset
    2: required list<TRow> rows
    3: optional list<TColumn> columns
    4: optional list<TEnColumn> enColumns
    5: optional binary compressorMap
}
```

where TEnColumn can have what is expected from a compressed column. The purpose for the compressorMap is to allow for a mix of compressed and uncompressed columns. For instance, if there are 3 columns C1, C2, and C3, where C1 is compressed while C2 and C3 are uncompressed, then only the first bit of the bitmap would be set while the others are not. It is important to note that a TRowSet can also contain uncompressed columns for a variety of reasons: the client did not request a compressor for the given type; the client requested a compressor but the Hiveserver2 was not configured with it. Ultimately, it is up to HiveServer2 to decide whether to compress a given column of a ResultSet.

TEnColumn would look like the following:

```
struct TEnColumn {
    1: required binary enData
    2: required binary nulls
    3: required Type type
    4: required i32 size
    5: required String compressorName
}
```

As part of TEnColumn, we are sending the compressed data, a set of nulls to address null values, the type of the column, the size and the compressor name. This is the minimum amount of data needed for the client to decompress a given ResultSet. In Thrift, we need to change TRowSet to accept compressed columns and include a new structure in Thrift that defines these compressed columns. Note that we have made it required to send compressorName for TEnColumn. This is so that whenever a column is compressed, the client can verify which compressor was used by the server and use the appropriate decompressor. We propose this TEnColumn as the absolute minimal administrative detail required for all compression techniques. Any given compression technique is free to pack its own details inside compressedData. For example, if a compression technique requires a look-up table in addition to the data, we suggest that the look-up table and the data values to be packed together into the compressedData.

Hive changes

New classes/interfaces created

To achieve the functionality of compressed columns, we introduced some new classes into Hive and also modified some existing ones. The details are below.

Column is the class that represents columns, and ColumnBasedSet is a list of Columns. To represent compressed columns, we have created EncodedColumnBasedSet.

```
public EncodedColumnBasedSet extends ColumnBasedSet {

}
```

A compressor in Hive needs to implement one common interface. This contract is adhered to by all compressors and helps Hive to identify them. We propose ColumnCompressor.java in org.apache.hive.service.cli.

The compressor would have two functions. One function, **isCompressable** will tell HiveServer2 if a given column can be compressed or not. This would save a call for HS2 in case the compressorSet cannot compress, for e.g, boolean columns as it does not have one. One can also write custom logic in this function to not compress some columns if it meets certain criteria (e.g if you don't want to compress columns with less than 100 rows, you can do it here.)

The second function is the actual compress function which accepts any column of class "Column" and returns a byte array.

```
@InterfaceAudience.Private
@InterfaceStability.Unstable
public interface ColumnCompressor {
    public boolean isCompressable(Column col);
    public byte[] compress(Column col);
}
```

Client/Server handshake

HiveServer2 may have multiple plugins for compressing ResultSets. A client can use one or more of them. There need to be a way for the client to inform the server which compressor it can support . To do so, we propose a JSON description of the form:

```
"INT_TYPE": {
 "vendor": "org.apache",
 "compressorSet": "HiveStock",
 "entryClass": "HiveStockInteger"
},
"DOUBLE_TYPE": {
 "vendor": "com.simba",
```

```
 "compressorSet": "compression",
 "entryClass": "doubleCompression"
}, . . .
```

Calling the plugin

When do you compress? Hive creates a RowSet to be sent to the client after all the columns have been formed. It is here that a compression library can be called to compress all the columns. Note that there are some types that the client may not want to be compressed at all. To address this, we allow for a mix of compressed and uncompressed columns (we allow both TColumns and TEnColumns in TRowSet.)

```
@Override
public TRowSet toTRowSet() {
 /*
    process the JSON string to find the vendor, compressorName and className.
    for each column in the list, if the type is part of the JSON string, compress
it else send as-is
    update the bitmap to reflect compression status
 */
}
```

Configuration - hive-site.xml

To activate/deactivate compression functionality for resultSets, the property **"hive.resultSet.compression.enabled"** can be set to true/false. This would be checked when a session is started and only if this property is set to true, compression would be used.

On the server side, a property, **"hive.resultSet.compressor.disable"** maintains a list of compressors that **shouldn't** be used for compression of ResultSets. This provides more control to the server side for activating/deactivating compressors.

Thus, to use a resultSet compressor, a client should mention the name as part of the JSON string and the compressor name shouldn't be a part of the disabled compressorList.

These are the two properties added to the hive-site.xml configuration file.

How do I write my own compressor/decompressor?

Good question! Here is a list of things you would need to do to build your own compressor/decompressor for Hive.

1.Provide an implementation of the interface ColumnCompressor

2.Configure the compressor into HiveServer2 via hive-site.xml.

3.Implement your decompressor into a client of your choice

4.Update the configuration handling in ThriftCLIService.java to read the property value you desire (e.g “com.custom.compressor”)

And that’s it! That’s how you can build your own compressor/decompressor for Apache Hive