SerDe

SerDe Overview

SerDe is short for Serializer/Deserializer. Hive uses the SerDe interface for IO. The interface handles both serialization and deserialization and also interpreting the results of serialization as individual fields for processing.

A SerDe allows Hive to read in data from a table, and write it back out to HDFS in any custom format. Anyone can write their own SerDe for their own data formats.

See Hive SerDe for an introduction to SerDes.

Built-in and Custom SerDes

The Hive SerDe library is in org.apache.hadoop.hive.serde2. (The old SerDe library in org.apache.hadoop.hive.serde is deprecated.)

Built-in SerDes

- Avro (Hive 0.9.1 and later)
- ORC (Hive 0.11 and later)
- RegEx
- Thrift
- Parquet (Hive 0.13 and later)
- CSV (Hive 0.14 and later)
- JsonSerDe (Hive 0.12 and later in hcatalog-core)

Note: For Hive releases prior to 0.12, Amazon provides a JSON SerDe available at `s3://elasticmapreduce/samples/hive-ads/libs/jsonserde.jar`.

Custom SerDes

For information about custom SerDes, see How to Write Your Own SerDe in the Developer Guide.

HiveQL for SerDes

For the HiveQL statements that specify SerDes and their properties, see Create Table (particularly Row Formats & SerDe) and Alter Table (Add SerDe Properties).

Input Processing

- Hive's execution engine (referred to as just engine henceforth) first uses the configured InputFormat to read in a record of data (the value object returned by the RecordReader of the InputFormat).
- The engine then invokes Serde.deserialize() to perform deserialization of the record. There is no real binding that the deserialized object returned by this method indeed be a fully deserialized one. For instance, in Hive there is a LazyStruct object which is used by the LazySimpleSerDe to represent the deserialized object. This object does not have the bytes deserialized up front but does at the point of access of a field.
- The engine also gets hold of the ObjectInspector to use by invoking Serde.getObjectInspector(). This has to be a subclass of structObjectInspector since a record representing a row of input data is essentially a struct type.
- The engine passes the deserialized object and the object inspector to all operators for their use in order to get the needed data from the record. The object inspector knows how to construct individual fields out of a deserialized record. For example, StructObjectInspector has a method called getStructFieldData() which returns a certain field in the record. This is the mechanism to access individual fields. For instance ExprNodeColumnEvaluator class which can extract a column from the input row uses this mechanism to get the real column object from the serialized row object. This real column object in turn can be a complex type (like a struct). To access sub fields in such complex typed objects, an operator would use the object inspector associated with that field (The top level StructObjectInspector for the row maintains a list of field level object inspectors which can be used to interpret individual fields).

For UDFs the new GenericUDF abstract class provides the ObjectInspector associated with the UDF arguments in the initialize() method. So the engine first initializes the UDF by calling this method. The UDF can then use these ObjectInspectors to interpret complex arguments (for simple arguments, the object handed to the udf is already the right primitive object like LongWritable/IntWritable etc).

Output Processing

Output is analogous to input. The engine passes the deserialized Object representing a record and the corresponding ObjectInspector to Serde.serialize(). In this context serialization means converting the record object to an object of the type expected by the OutputFormat which will be used to perform the write. To perform this conversion, the serialize() method can make use of the passed ObjectInspector to get the individual fields in the record in order to convert the record to the appropriate type.

Additional Notes

- The owner of an object (either a row, a column, a sub field of a column, or the return value of a UDF) is the code that creates it, and the life time of an object expires when the corresponding object for the next row is created. That means several things:
    - We should not directly cache any object. In both group-by and join, we copy the object and then put it into a HashMap.
    - SerDe, UDF, etc can reuse the same object for the same column in different rows. That means we can get rid of most of the object creations in the data pipeline, which is a huge performance boost.

- Settable ObjectInspectors (for write and object creation).
    - ObjectInspector allows us to "get" fields, but not "set" fields or "create" objects.
    - Settable ObjectInspectors allows that.
    - We can convert an object with JavaIntObjectInspector to an object with WritableIntObjectInspector (which is, from Integer to IntWritable) easily with the help of Settable ObjectInspectors.
    - In UDFs (non-GenericUDFs), we use Java reflection to get the type of the parameters/return values of a function (like IntWritable in case of UDFOPPlus), and then infer the ObjectInspector for that using ObjectInspectorUtils.getStandardObjectInspectors.
    - Given the ObjectInspector of an Object that is passed to a UDF, and the ObjectInspector of the type of the parameter of the UDF, we will construct a ObjectInspectorConverter, which uses the SettableObjectInspector interface to convert the object. The converters are called in GenericUDF and GenericUDAF.

In short, Hive will automatically convert objects so that Integer will be converted to IntWritable (and vice versa) if needed. This allows people without Hadoop knowledge to use Java primitive classes (Integer, etc), while hadoop users/experts can use IntWritable which is more efficient.

Between map and reduce, Hive uses LazyBinarySerDe and BinarySortableSerDe 's serialize methods. SerDe can serialize an object that is created by another serde, using ObjectInspector.