Spark-HBase Connector Design

Spark-HBase connector is provides an easy way for user to retrieve data from HBase inside of Spark. However, current implementation is built on top of TableInputFormat, and may not fit to different scenarios due to the limitation of TableInputFormat, which is initially built for tradtional MapReduce tasks instead of Spark.

This document provide an alternate design for Spark-HBase connector and provide users an option to choose different implementation based on their specific requirements or use cases.

Repo:
https://github.com/zhzhan/shc

Rationale

The current implementation is based on TableInputFormat, which has a lot of unnecessary limitations. For example, it only accepts one Scan at a time. To walkaround this limitation, multiple tasks are constructed with each TableInputFormat having one Scans. As a result, the number of Spark tasks is equal to the number of non-overlapping ranges in user query.

There are limited number of spark executors, and each of them has a fixed number of cores. When the number of tasks is larger than the Spark cores, some of tasks have to be delayed until there are cores available. To make it worse, the leftover tasks later may have to be scheduled to executors (which have finish the earlier tasks) without data locality.

In the meantime, big task number also increase the system task sedes and scheduling overhead.

In addition, the TableInputFormat does not provide BulkGet. To walkaround this issue, currently all gets are performed at the driver side, which means the driver has to collect the data and redistribute them to executors. It may cause huge network overhead, and single point of performance bottleneck. More seriously, spark driver typically does not take heavy tasks, because it is the central point of the whole system, and its failure will cause the whole job not recoverable.

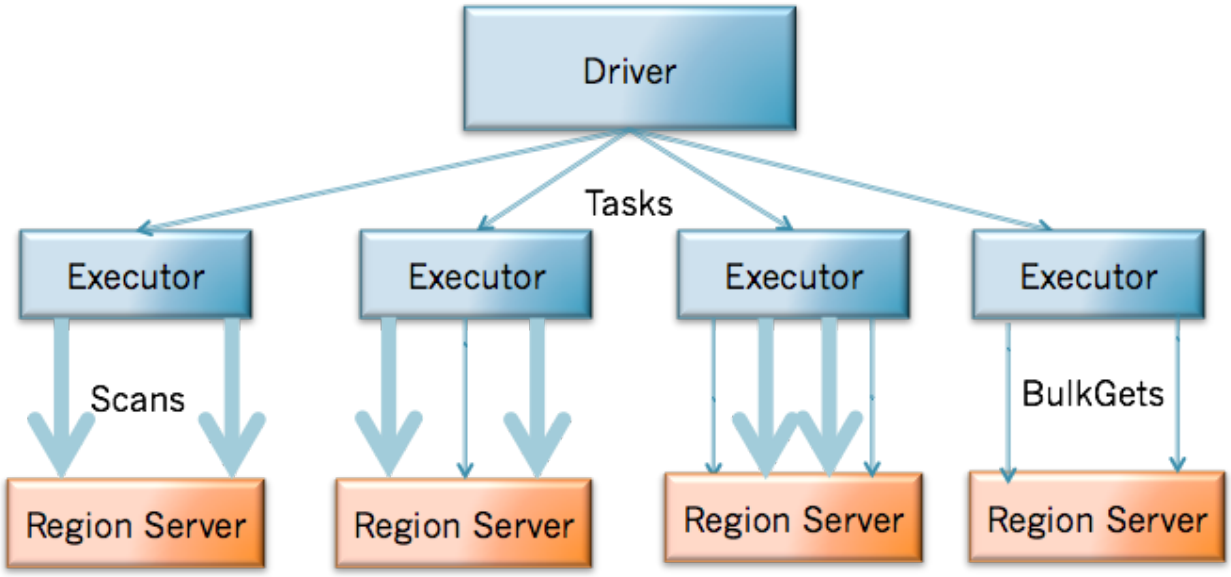
To overcome these potential bottleneck and provide users an alternate option, we propose a new architecture and implementation as a plugin so that users can choose which architecture to use based on their specific use cases.

Architecture
The connector treats the Scan and Get in similar way, and both actions will be performed on the executors. On the high level, the driver processes the query, aggregates scans/gets by the region server, and generates tasks per region server. Each task may consists of multiple scans

and BulkGets, and the data requests by one tasks can be retrieved from one specific region server. The locality preference of each task is set to the corresponding region server host. The tasks are sent to the executors co-located with the region server, and are performed in parallel in executors in order to achieve better data locality and concurrency.

Note that by aggregation the number of tasks is no larger than the number of region servers, indicating that all tasks can be scheduled to executors and achieve good data locality.

In addition, the BulkGets are also performed in the executors, and the driver is not involved in the real job execution except constructing tasks, which avoid the driver as the single bottleneck, and crashing driver accidently.



Catalog
For each table, a catalog has to be provided, which includes the row key, and the columns with data type with predefined column families, and defines the mapping between hbase column and table schema. The catalog is user defined json format.

Datatype conversion
Java primitive types is supported. In the future, other data types will be supported, which relies on user specified sedes. Take avro as an example. User defined sedes will be responsible to convert byte array to avro object, and connector will be responsible to convert avro object to catalyst supported datatypes.

Note that if user want dataframe to only handle byte array, the binary type can be specified. Then user can get the catalyst row with each column as a byte array. User can further deserialize it with customized deserializer, or operate on the RDD of the data frame directly.

Solve the conflicts of Catalyst datatype order and HBase byte array order
The library automatically handle the orderness between the conflicts between the java data type (Short, Integer, Long, Float, Double, String, etc) and HBase bytearray order, which means as long as the format saved in HBase is consistent with the Java native data format, the libary will be able to take care of pruning and comparison instead of relying on a specific import tools.

Data locality
When the spark worker node co-located with hbase region servers, data locality is achieved by identifying the region server location, and co-locate the executor with the region server. Each executor will only perform Scan/BulkGet on the part of the data that co-locates on the same host. Because the number of tasks is no larger than the region server, there is a good chance to achieve data locality.

Predicate pushdown
The lib use existing standard HBase filter provided by HBase and does not operate on the coprocessor.

Partition Pruning
By extracting the row key from the predicates, we split the scan/BulkGet into multiple non-overlapping ranges, only the region servers that have the requested data will perform scans/BulkGets. Currently, the partition pruning is performed on the first dimension of the row keys. Note that the WHERE conditions need to be defined carefully. Otherwise, the result scanning may includes a region larger than user expected. For example, following condition will result in a full scan of the table (rowkey1 is the first dimension of the rowkey, and column is a regular hbase column). WHERE rowkey1 > "abc" OR column = "xyz"

Scan and BulkGet
Both are exposed to users by specifying WHERE CLAUSE, e.g., where column > x and column < y for scan and where column = x for get. All the operations are performed in the executors, and driver only constructs these operations. Internally we will convert them to scan or get or combination of both, which return Iterator[Row] to catalyst engine.

Dataframe Write Support
The connector supports data frame persist to the HBase table directly.

Composite Key Support
The connector allows the HBase table consisting of mulitple keys. Currently, the partition prunning happens on the first dimension only.

Application API/Usage

Define Catalog

```
def catalog = s"""{
       |"table":{"namespace":"default", "name":"table1"},
       |"rowkey":"key",
       |"columns":{
         |"col0":{"cf":"rowkey", "col":"key", "type":"string"},
         |"col1":{"cf":"cf1", "col":"col1", "type":"boolean"},
         |"col2":{"cf":"cf2", "col":"col2", "type":"double"},
         |"col3":{"cf":"cf3", "col":"col3", "type":"float"},
         |"col4":{"cf":"cf4", "col":"col4", "type":"int"},
         |"col5":{"cf":"cf5", "col":"col5", "type":"bigint"},
         |"col6":{"cf":"cf6", "col":"col6", "type":"smallint"},
         |"col7":{"cf":"cf7", "col":"col7", "type":"string"},
         |"col8":{"cf":"cf8", "col":"col8", "type":"tinyint"}
       |}
     |}""".stripMargin
```

The above defines a schema for a HBase table with name as table1, row key as key and a number of columns (col1-col8). Note that the rowkey also has to be defined in details as a column (col0), which has a specific cf (rowkey).

Populate Data Using Dataframe Write

```
sc.parallelize(data).toDF.write.options(
  Map(HBaseTableCatalog.tableCatalog -> catalog, HBaseTableCatalog.newTable -> "5"))
  .format("org.apache.spark.sql.execution.datasources.hbase")
  .save()
```

Given a data frame with specified schema, above will create an HBase table with 5 regions and save the data frame inside. Note that if HBaseTableCatalog.newTable is not specified, the table has to be pre-created.

Perform data frame operation on top of HBase table

```
def withCatalog(cat: String): DataFrame = {
  sqlContext
  .read
  .options(Map(HBaseTableCatalog.tableCatalog->cat))
  .format("org.apache.spark.sql.execution.datasources.hbase")
  .load()
```

```
}
```

Complicated query

```
val df = withCatalog(catalog)
val s = df.filter((($"col0" <= "row050" && $"col0" > "row040") ||
  $"col0" === "row005" ||
  $"col0" === "row020" ||
  $"col0" ===  "r20" ||
  $"col0" <= "row005") &&
  ($"col4" === 1 ||
  $"col4" === 42))
  .select("col0", "col1", "col4")
s.show
```

SQL support

```
// Load the dataframe
val df = withCatalog(catalog)
//SQL example
df.registerTempTable("table")
sqlContext.sql("select count(col1) from table").show
```

TODO

Following illustrates our next step, which includes complex data types, support of customized sedes and avro.

```
val complex = s"""MAP<int, struct<varchar:string>>"""
val schema =
  s"""{"namespace": "example.avro",
     |   "type": "record",      "name": "User",
     |    "fields": [      {"name": "name", "type": "string"},
     |      {"name": "favorite_number",  "type": ["int", "null"]},
     |       {"name": "favorite_color", "type": ["string", "null"]}      ]    }""".stripMargin
val catalog = s"""{
       |"table":{"namespace":"default", "name":"htable"},
       |"rowkey":"key1:key2",
       |"columns":{
         |"col1":{"cf":"rowkey", "col":"key1", "type":"binary"},
         |"col2":{"cf":"rowkey", "col":"key2", "type":"double"},
         |"col3":{"cf":"cf1", "col":"col1", "avro":"schema1"},
```

```
          |"col4":{"cf":"cf1", "col":"col2", "type":"string"},
          |"col5":{"cf":"cf1", "col":"col3", "type":"double",
"sedes":"org.apache.spark.sql.execution.datasources.hbase.DoubleSedes"},
          |"col6":{"cf":"cf1", "col":"col4", "type":"$complex"}
        |}
      |}""".stripMargin

val df = sqlContext.read.options(Map("schema1"->schema,
HBaseTableCatalog.tableCatalog->catalog)).format("org.apache.spark.sql.execution.datasource
s.hbase").load()
df.write.options(Map("schema1"->schema,
HBaseTableCatalog.tableCatalog->catalog)).format("org.apache.spark.sql.execution.datasource
s.hbase").save()
```