Proposal of Spark SQL on HBase

Yan Zhou

July 14, 2015

Version 2.2

Contents

Modification History

Version 2.1: Coprocessor

Overview

Apache HBase is a distributed Key-Value store of data on HDFS. It is modeled after Google's Big Table, and provides APIs to query the data. The data is organized, partitioned and distributed by its "row keys". Per partition, the data is further physically partitioned by "column families" that specify collections of "columns" of data. The data model is for wide and sparse tables where columns are dynamic and may well be sparse.

Although HBase is a very useful big data store, its access mechanism is very primitive and only through client-side APIs, Map/Reduce interfaces and interactive shells. SQL accesses to HBase data are available through Map/Reduce or interfaces mechanisms such as Apache Hive and Impala, or some "native" SQL technologies like Apache Phoenix. While the former is usually cheaper to implement and use, their latencies and efficiencies often cannot compare favorably with the latter and are often suitable only for offline analysis. The latter category, in contrast, often performs better and qualifies more as online engines; they are often on top of purpose-built execution engines.

Currently Spark supports queries against HBase data through HBase's Map/Reduce interface (i.e., TableInputFormat). SparkSQL supports use of Hive data, which theoretically should be able to support HBase data access, out-of-box, through HBase's Map/Reduce interface and therefore falls into the first category of the "SQL on HBase" technologies.

We believe, as a unified big data processing engine, Spark is in good position to provide better HBase support.

1. Code Structure

The source files will be in the sql/hbase subdirectory of the Spark source tree that contains subdirectories of utl, dsl, api, catalyst and execution, whose purposes will be explained in later sections.

2. Supported HBase Version

HBase 0.98 will be supported.

3. APIs

Python APIs will be provided.

4. Command Line Interface

An (enhanced) command line interface (CLI) will be provided to support the new DDL/DML commands introduced in this project.

5. Metadata Persistence

Table meta data will be stored in a HBase table named “SPARK_SQL_HBASE_TABLE”, of a single column family named “CF”. Each SQL table will use a single row in the HBase table, referred as the “meta table” hereafter. And each column will store the name and type encoding of a column of the SQL table.

Metadata related codes are in the “meta” subdirectory

6. Interactive shell and DSL

The interactive shell is essentially a combo of Spark shell and HBase shell. The Spark shell will provide the same functionalities as what Spark and SparkSQL currently do. The functionalities from the HBase shell will add HBase-specific ones to the Spark shell. This will facilitate seasoned HBase users’ and admins’ transition to the Spark world. Furthermore, Spark/Scala power can be applied to results from HBase server. For this purpose, a DSL of existing HBase shell commands will be created. The output from this DSL can then be at hand for further processing, which is not possible with HBase shell by itself. An example is as follows:

spark-hbase> scan ‘mytable’

res0: ((String, String), Seq[(String, String)]) =

ROW COLUMN+CELL

Row1 column=cf1.c1,timestamp=12345678,value=v1

Row2 column=cf2.c2,timestamp=12345679,value=v2

spark-hbase>res0._2.filter(_._2.equals(Row2))

res1: Seq[(String, String)] =List((Row2, column=cf2.c2,timestamp=12345679,value=v2))

Codes for DSL will be put in the "dsl" subdirectory.

Note that this feature is not to be confused with SchemaRDD's DSL, which is still supported for HBase-based SchemaRDDs.

The functionalities are primarily for use convenience, and could be put in a separate "contrib" source subdirectory instead of the main source tree.

This feature is not supported in the first version.

7. RDD and Partitions

A new type of relation, HBaseRelation, will be introduced to work as a bridge between SparkSQL physical runtime and HBase-specific data access mechanism. Its functionalities include management of HBase configuration and connections, and providing various mapping and conversion mechanisms and utilities, including the (logical) table schema, key/column mappings, key composition and extraction convenience methods, …, etc. The connection at the client side will be made by the catalog during its relation lookup process and kept with the HBaseRelation instance. At the executor side, the connections will be made either by each task individually and closed at its finish, or by the external resource pool. HBaseRelation will be used by HBaseSQLReaderRDD, which supports filtered scan and application of HBase coprocessor and will be used by a new data source node in the physical plan, HBaseSQLTableScan. It will also be contained in a new ShuffledRDD, HBaseLoadRDD, for the reducer work by the bulk load operator, LoadIntoHBaseTable. Note that the HBaseRelation must be serializable for usability by the slaves.

Partitions of HBase data are through the HBase regions. Specifically, RDD's getPartitions will return the range partitions as embodied by HBase regions; RDD's getPreferredLocations will return the hosts of HBase's region servers if HBase and Spark are collocated on the same set of machines.

HTable's methods of getStartKeys and getRegionLocations can be used to fetch the region information from HBase server.

Co-located execution thus attempted can minimize network traffic. In future, when Spark supports long-running services, and either Spark Executor or HBase region server can be "enginized", even the (local) data transfer between the region server and the Spark executor could be avoided.

8. Design Principles

There are a few design principles we would like to follow to maximize the user convenience on Spark as a unified distributed execution engine.

1) It is intended to have least intrusive code impact on other modules outside of this subproject. This principle includes
    a. Maximize the support of SparkSQL's functionalities. The few remaining areas where differences have to be present, most noticeably DDL, will be implemented in an extension of SqlContext, called HBaseSQLContext, where differentiation will be implemented through extended parser, physical optimizer and execution engine specific to HBase-based tables. Another advantage of this compatibility principle is to allow for queries against a mix of HBase tables and others.
    b. On the development side, every effort will be made to isolate changes to the hbase subdirectories. If needed, code copy/paste could be applied if access restriction forbids class inheritance or method overriding before a resolution can be reached on potential changes in the parent project, SparkSQL.
    c. Existing coding patterns/models/paradigms are to be honored to the maximum possible degree.
2) Access to the HBase data source is provided through an implementation of a new Spark SQL 1.3 "foreign data source interface".

9. Deployment

It is required that all Spark slave machines be configured as HBase, and implicitly Zookeeper, clients. It is preferable that the Spark and HBase cluster are co-located on the same set of physical or virtual boxes, but it is not actually a must.

Coprocessor- and custom-filter-related HBase configurations and necessary jars containing corresponding logics from SparkSQL will be deployed to the HBase cluster.

Specifically, the hbase-site.xml will need to be added the following four lines:

```
<property>
  <name>hbase.coprocessor.user.region.classes</name>
  <value>org.apache.spark.sql.hbase.CheckDirEndPointImpl</value>
</property>
```

In the hbase-env.sh script, the HBASE_CLASSPATH need to add the Spark jar and the spark-hbase jar of this product.

10. Configuration

HBase configuration will be through either the Spark configurations with the conventional "spark.sql.hbase" prefix.

Currently, there are four supported configuration flags:

1. spark.sql.hbase.partition.expiration specifies the expiration time (in seconds) of the cached HBase table region information. The default is 600 for 10 minutes.
2. spark.sql.hbase.scanner.fetchsize specifies the HBase scanner fetch size and defaults to 1000.
3. spark.sql.hbase.coprocessor is a Boolean to switch on/off the use of coprocessors.
4. spark.sql.hbase.customfilter is a Boolean to switch on/off the use of custom filters.

Explanations of the configuration flags are also included in their respective related sections.

11. HBase Connection Handling

Connection to meta table store will be created and cached in the HBase clients.

For HBase and Zookeeper connections embodied in the HTable instance from the Spark executors, since currently there is no support for long-living external resources by the Spark executors, it is recommended here to add a new "external resources" to Spark core's Executor class, and a corresponding addExternalResource method to SparkContext. Existing "Files" and "Jars" can be absorbed into this new "external resource" class. Reflection can be used to construct external resources per client's desire. At the Executor side, an asynchronous thread will close, if so supported and through an interface method, an external resource that has been inactive for a configurable expiration time that defaults to, say, 2 minutes. At Executor's exit, all Connections will be closed.

HBase's Configuration, which implicitly incorporates HBase and Zookeeper connections, could be one type of the "external resource" and will support interface methods of "start" and "stop".

Moreover, a HTablePool is another external resource that can serve as a "native" HBase connection pool.

A JIRA, SPARK-3306, has been filed for use of the external resource by executors. Due to its more general nature, it is filed as a separate JIRA.

12. HBase Scanner Caching and Partition caching

A default value of 1000 is set for a configurable setting of the HBase scanner caching, which is disabled by HBase scanner default. It is configuration through a configuration variable of spark.sql.hbase.scanner.fetchsize.

This value is configurable in the hbase-site.xml file through the “hbase.client.scanner.caching” property. Unlike HBase, though, the default value is 100 not 1; and we do not support the configuration on a table-wide basis.

In the first release, the value is not configurable and fixed at the default of 100.

On the other hand, the HBase region partition information is cached for a default of 10 minutes, and is configurable through a new configuration variable of “spark.sql.hbase.partition.expiration”.

13. Row Key Composition

HBase row keys will be composed in the way of Big Endian, for processing efficiency. Keys, or key components, of the STRING type are marked with a NULL terminator.

14. Partition Pruning and Predicate Pushdown

For organized, distributed data sets , it is important to have a good and efficient partition pruner and predicate down pusher to take the maximum advantage of the data organization and the processing capabilities of the underlying storage. For this purpose, a technique based upon partial evaluation is utilized to (near-)precisely and efficiently generate the pruned partitions and partition-specific predicates to be pushed down to HBase as its filters.

15. Utilities

It is conceivable now that the “utilities” will include Key composer and decomposer, data type converters between HBase’s byte[] and various SQL data types.

The codes will be in the “utl” subdirectory.

16. SQL Support

Queries and data types will be the same as what SparkSQL supports. The differences will be in DDL and DML.

16.1 DDL

Note that all DDL statements only affect the logical SQL table and not the physical tables.

16.1.1 CREATE TABLE

A “create table” statement will be of the form of:

CREATE TABLE table_name (col1 TYPE1, col2 TYPE2, …, PRIMARY KEY (col7, col1, col3))

MAPPED BY (hbase_tablename, COLS=[col2=cf1.cq11, col4=cf1.cq12, col5=cf2.cq21, col6=cf2.cq22]) [IN StringFormat]

A SQL table on HBASE is basically a logical table mapped to a HBase table. This mapping can be many-to-one to support “schema-on-read” for SQL access to HBase data.

The “hbase_table_name” denotes the HBase table; the “primary key” constraint denotes the HBase row key composition of columns; “col2=cf1.cq1” denotes the mapping of the second column to the HBase tables column qualifier of “cq1” of column family “cf1”. The table and the column families specified have to exist in HBase for the CREATE TABLE statement to succeed. In addition, the columns in the primary key cannot be mapped to another column family/column qualifier combo. Other normal SQL sanity checks, such as uniqueness of logical columns, will be applied as well.

For the table created by general HBase command, it could also be supported as long as the native type of its key is String. For that case, the key word “IN StringFormat” should be included at the end.

16.1.2 DROP TABLE

A “drop table” statement is of the form of

DROP TABLE table_name

This will not delete the HBase table the SQL table maps to, but just deletes the SQL table with its schema.

16.1.3 ALTER TABLE

ALTER TABLE table_name DROP column

Drops an existing column from the SQL table.

ALTER TABLE table_name ADD col1 TYPE1 MAPPED BY (col1 = cf.cq)

Adds a new column that is mapped to existing column family “cf” and column qualifier “cq”.

ALTER TABLE does not support addition or deletion of components in the composite row key

16.2 DML

16.2.1 INSERT

The syntax remains the same as SchemaRDD’s. One constraint is that all columns in the HBASE key must be present for insertion to succeed.

Normal SQL sanity checks for INSERT, such as uniqueness of logical columns, will be applied.

There are two types of inserts. The first has the following syntax:

INSERT INTO TABLE table_name VALUES (col1_value, col2_value, …)

While the second has

INSERT INTO TABLE table1_name SELECT … FROM table2_name

16.2.2 Bulk Loading

A “LOAD DATA [PARALLEL] INPATH filePath INTO TABLE tableName [FIELDS TERMINATED BY char]”, similar to Hive’s, will be used to invoke a bulk loading into existing HBase tables.

HFileOutputFormat will be used to i) write out data in HBase’s internal storage format; ii) set up a Spark Partitioner similar to Hadoop’s TotalOrderPartitioner to partition the map output into disjoint ranges of the HBase key space corresponding to key ranges of the HBase regions. (Spark’s RangePartitioner can’t be utilized for its sampling-based partition generation.) The configureIncrementalLoad convenience method of HBase could be consulted with for this purpose. This is essentially a M/R process .

Eventually the “reducers” will contact HBase to ship the files in the HBase internal format to HBase. It will have to deal with changed regions if the HBase table is not pre-split. For this, the “completebulkload” method of HFileOutputFormat might be utilized.

An optimization is to schedule the “reducers” on the same servers where the region servers run to minimize the data transfers across network at query time. Spark schedules reducers randomly with no setting of “preferredLocations” in the ShuffleRDD. Accordingly, an extended ShuffledRDD with an overridden “getPreferredLocations” method, HBaseLoadRDD, will take a range partitioner according to HBase regions, and will be scheduled collocated with the corresponding HBase region servers.

A “parallel” option will merge the “incremental loading” phase into the HFile generation phase. Conceivably it will perform better, particularly for non-presplit tables.

16.3 Miscellaneous Commands

Additionally, “Show TABLES” and “DESCRIBE *table*name” commands are supported for sake of convenience.

17. New files, classes and functionalities

17.1 HBaseCatalog extends Catalog

In addition to the interface methods of Catalog, supports create/drop table, stores a column mapping between the SparkSQL table and the physical HBase table, and the client connection to the HBase table storing the metadata.

17.2 HBaseSqlParser extends SqlParser

Supports DDL/DML capabilities on tables in HBase. One caveat is that this HBase Table-specific parser introduces extra set of key words beyond what Spark SQL has, that will cause parsing failure if any of them are attempted to be used for any other purpose. This set of new key words includes, exhaustively, “add”, “alter”, “boolean”, “ byte”, “cols”, “create”, “data”, “describe”, “drop”, “exists”, “fields”, “float”, “inpath”, “int”, “integer”, “key”, “load”, “local”, “long”, “mapped”, “primary”, “short”, “show”, “values”, and “terminated”.

17.3 HBaseSQLContext extends SQLContext

Contains HBaseSqlParser;

Overrides the “sqlParser” value to graft an enhanced SQL parser with HBase-specific DDL/DML;

Assigns “extraStrategies” with HBase-specific strategies including the scan and “insert … into … values”;

There is no support of the (implicit) createSchemaRDD from a case class. This is due to the complexity to specify a new SQL table on a physical HBase table, mainly the key/column mapping;

There is no support of createTable using the schema of a case class, for the same sake as above.

17.4 HBasePartition extends Partition

Contains a HBase row key range and a server host.

17.5 HBaseSQLReaderRDD extends RDD

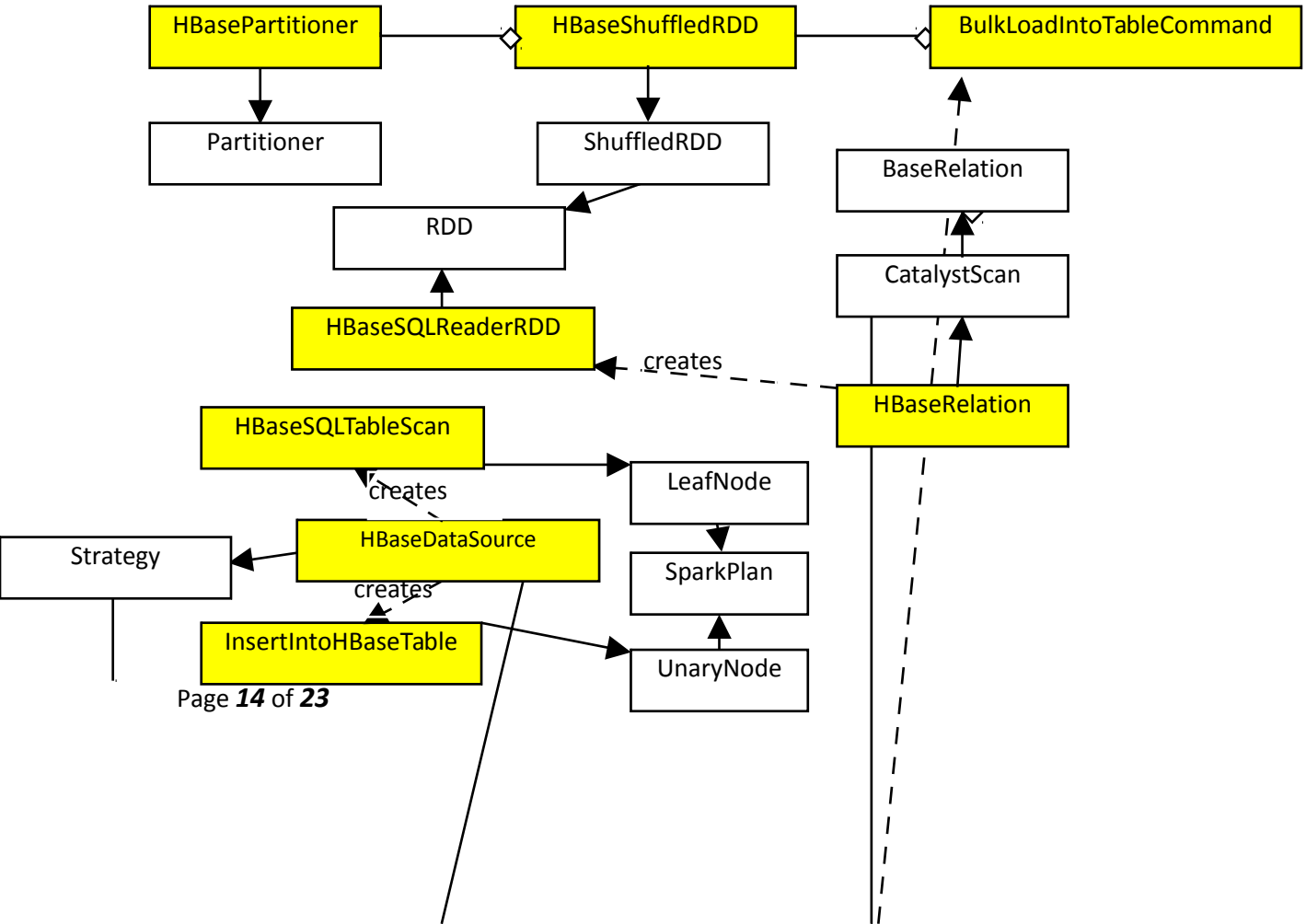
Sets up HBase Filters and EndPoint coprocessors properly;

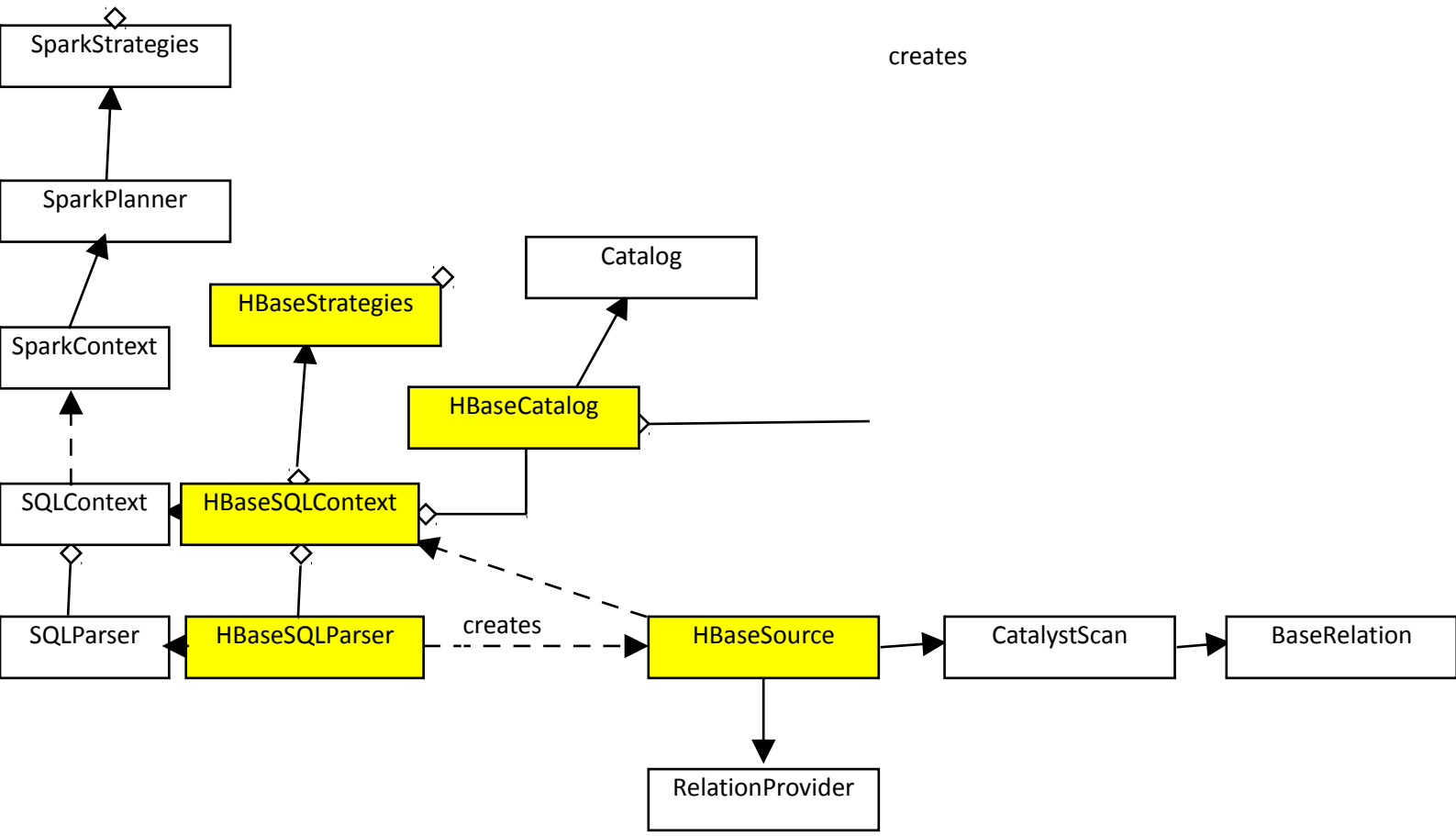
Instantiates a GET for queries on “fully qualified rows, or a SCAN for other queries;

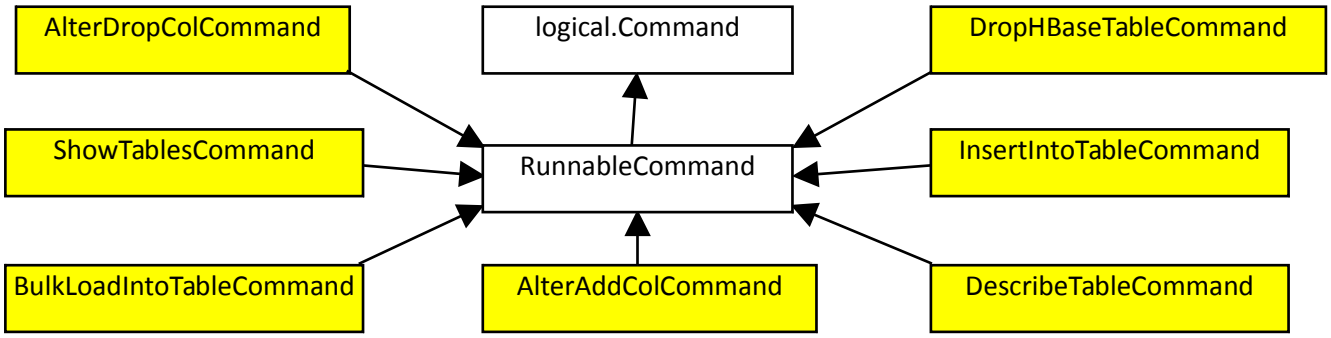
Fetches raw rows, possibly filtered and “coprocessed”, from HBase.

17.6 Class Diagram

Yellow-colored classes are new and specific for Spark SQL on HBase; while the white-colored ones are of Spark SQL.







18. Data Frame

Data Frame's functionalities will be supported. An example statement is as follows:

```
val  hbaseContext =  new HBaseSQLContext(sc)
hbaseContext.read.format(“org.apache.spark.sql.hbase.HBaseSource”).options(
   Map("namespace" -> "", "tableName" -> "people", "hbaseTableName" -> "people_table",
     "colsSeq" -> "name,age,id,address",
     "keyCols" -> "id,integer",
     "nonKeyCols" ->
       "name,string,cf1,cq_name;age,integer,cf1,cq_age;address,string,cf2,cq_address")).load
hbaseContext.sql("Select  `personal_data:name`, `personal_data:identification` as b, `personal_data


hbaseContext.sql("Select  name,  id, address  from people").collect.foreach(println)
```

There will be a few potentially subtle difference worth of caution. In particular, the methods of “registerAsTable” will not “register” a HBase-based table but a usual SparkSQL table. On the other hand, “insertInto” and “saveAsTable method will insert data into an existing SQL table on top of a HBase table.

19. Java Support

The HBaseSQLContext can be created as follows with sc being an existing JavaSparkContext:

```
HBaseSQLContext  hsqlcxt = new org.apache.spark.sql.hbase.HBaseSQLContext(sc);
```

20. Python Support

20.1 Python Shell

The SPARK_HOME environment needs to be set. Then go to the root of the Spark-SQL-on-HBase installation, and issue, for instance,

%bin/pyspark-hbase --master=spark://*your-spark-master-ip*

20.2 Python Script

To build a HBaseSQLContext, use the following Python statements:

```
from pyspark_hbase.sql import HBaseSQLContext, context

sc = SparkContext(……)
```

context.register(sc)

hsqlContext=HBaseSQLContext(sc)

In addition, set your PYTHONPATH environmental variable to the “python” directory under the of the Spark-SQL-on-HBase installation.

21. Coprocessor

“Region coprocessors” will be utilized for sub-plans of the optimized Spark SQL physical plan formed from the HBaseSQLTableScan up until just under the Exchange (or Limit) node if possible. The rationale behind this is to push down as many Spark SQL operations as possible; while leaving shuffling and the operations after it to the Spark SQL execution engine. This consideration also carries an advantage of separating Spark’s memory management from that of HBase Region server.

The coprocessor processing will be embodied in the coprocessor subplan which, in turn, will be constructed, serialized and passed over from Spark slaves. And it requires deployment of a jar file of the coprocessor logic to the HBase region server nodes, and necessary changes to the hbase-site.xml configuration file.

21.1 Availability and Loading of Coprocessor

Loading of coprocessor through the HBase table descriptor in the HBase region servers will be supported. It will be loaded when the Spark SQL driver connects or reconnects to the HBase.

The HBase table descriptor will be consulted as whether the coprocessor class, named “org.apache.spark.sql.hbase.coprocessor.SparkSQLRegionObserver”, is loaded. If not, the execution will fall back to the one without support of the coprocessor. A new Boolean variable , “coprocessor”, will be added to the HBaseRelation class to keep track of the fact of the support of coprocessor.

A configuration flag of “spark.sql.hbase.coprocessor”, defaulting to “true”, switches on/off the coprocessor processing.

21.2 Coprocessor Sub-Plan

If the coprocessor is supported by the Spark SQL table on HBase, an extra physical plan generation step will be performed as part of the “prepareForExecution” step that will be overridden and

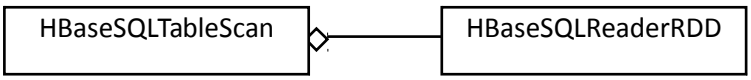
appended with a “coprocessor planning” step after the original “Add exchange” step so that the sub-plan will be plucked from just under the Exchange node if present. The original physical plan will be modified to replace the coprocessor sub-plan with a new instance of the HBaseSQLTableScan class whose “output “ is that of the coprocessor sub-plan.

The new instance of the HBaseSQLTableScan class contains an instance of a new “HBasePostCoprocessorSQLReaderRDD” class of a non-transient instance variable for the plucked-off sub-plan” and whose “compute” method will perform i) serialize the coprocessor sub-plan; ii) construct a HBase scan with the same key range, filters and projections as by the original HBaseSQLTableScan. (If the original HBaseSQLTableScan only creates HBase Get or Get list, the coprocessor won’t be invoked;) iii) call the Scan’s setAttribute method to pass over the serialized coprocessor sub-plan.

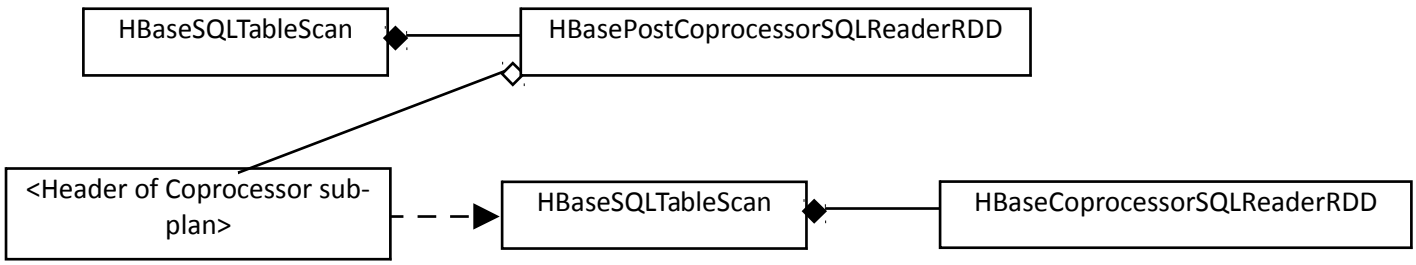
The plucked-off sub-plan will have its leaf, which is a HBaseSQLTableScan instance, to be replaced with another HBaseSQLTableScan instance containing an instance of a new HBaseCoprocessorSQLReaderRDD class whose “codegenEnabled “ instance variable will be set and the “otherFilter” instance variable be computed subsequently.

Graphically these classes and their relationships can be summarized as follows:

For the original physical plan:



For the post-coprocessor and coprocessor physical plans:



21.3 The coprocessor execution by the Region Server

The new “HBaseCoprocessorSQLReaderRDD” class will be introduced to handle the scan operation by the coprocessor in the region servers. It will i) replace the original HBaseSQLReaderRDD node in the coprocessor sub-plan; ii) have a transient instance variable for the original scanner; iii) have a non-transient instance variable “otherFilter” denoting the “non-pushdownable” filters that will be

serialized by and passed over from the Spark SQL slave; and iii) run the “other filter” on the after row returned from the original scanner.

The “SparkSQLRegionObserver”, extending HBase’s “BaseRegionObserver” class, works as the coprocessor for the coprocessor sub-plan execution. For this, its implementation of the (overridden) doPostScannerOpen will i) get the serialized coprocessor sub-plan from the original scan’s getAttribute method; ii) instantiate a coprocessor sub-plan from deserialization of the serialized coprocessor subplan; iii) create a instance of the new “SparkSQLRegionScanner” class as an implementation of the HBase’s regionserver.RegionScanner interface; iv) sets up the original scan to the leaf of the sub-plan, which is an instance of the new “coprocHBaseSQLReaderRDD” class.

The “SparkSQLRegionObserver” will necessarily be a Java class and be deployed to the HBase region servers.

The “next” method of the new “SparkSQLRegionScanner” class will simply iterate the result as returned by the subplan’s “execute” method.

The “code generation” will be supported by the coprocessor. But the unsafe-related execution is not supported by the coprocessor. For one, the memory manager is not used by the coprocessor.

21.4 HBaseRelation caching and HTablePool

Caching of HBaseRelation in the coprocessor will be supported through a singleton Java class static variable of the “SparkSQLRegionObserver” class for which synchronized access will be necessary. The HTablePool will be supported as well.

After the Spark SQL metadata change, the notification will be piggybacked from the first query to trigger a refresh/rebuild process of the cached HBaseRelation instance.

In the first development phase, however, the HBaseRelation could be serialized and sent over from the Spark slaves.

21.5 Phases of Development

Initially joins will be performed outside of coprocessor. In the future, it may be added per user requirements.

22. Custom Filters

22.1 Row skips from Filters on Non-leading Dimension Key

It'd be beneficial to skip the rows of the same value of the leading dimension key(s) to the next possible value of the leading dimension key(s) if the current dimension key and its subsequent value cannot meet the filter range predicate. The Filter's getNextKeyHint can be used for this purpose. A precondition for the applicability of this technique is that the filtering predicate is either equality or range comparison.

The "partial evaluation" technique can be utilized in a progressively manner from the leading dimension down to trailing ones to find qualified "critical point ranges" based upon the current (unqualified) value of the leading key(s) under scan. The result of the partial evaluation can't be TRUE. Then the getNextKeyHint can work at two levels.

First, if the current qualified "critical point ranges" have not been exhausted, but the current qualified "critical point range" is exhausted, set the next key hint to the next qualified "critical point range".

Second if the current qualified "critical point range" and the qualified "critical point ranges" are both exhausted, find the next qualified "critical point range" from the leading (i.e., the more significant) dimension, or, if the leading dimension's qualified "critical point range" and qualified "critical point ranges" are exhausted, proceed progressively to its leading dimension.

If the partial evaluation result for a particular dimension is FALSE, the qualified "critical point ranges" are considered to be empty.

If the partial evaluation result for a particular dimension is MAYBE, but the qualified "critical point ranges" of the dimension is empty, which could occur when there is no range/equality predicate on the dimension, the qualified "critical point range" is considered to be the full range subject to possible partition range limits for the non-leading dimension, and the step of increment has to be the minimal of "one".

Note that for any dimension in a partition, any qualified "critical point range" from a partial evaluation must be a subset and sub-range of the qualified "critical point ranges" as partially evaluated on the partition. Considering the potential costs of partial evaluation on every unqualified rows, it is beneficial to first generate qualified, partition-wide "critical point ranges" for each dimension and for the composite dimension. Then the scan will only be from each qualified "critical point ranges" for the composite dimension; and partial evaluation for each unqualified rows will generate, for each dimension from the next-to-last to the first (most significant) in the order of significance if the dimension has non-empty qualified "critical point ranges" in the partition, dimension-specific qualified "critical point ranges". The partial evaluation will be on the partial row of the unqualified row with the values of key columns from the first dimension down to the dimension being considered.

A configuration flag of "spark.sql.hbase.customfilter", defaulting to "true", switches on/off the use of the custom filter

.

22.2 Filter on any portion of the Row Key

Currently only PrefixFilter filter can be used for filtering on a portion of a row key. Conceivably it'd be beneficial to use a filter that can filter based upon any portion of the row key, corresponding to any dimension of the composite primary key.

If the filtering predicate is range comparison or equality , this technique is actually consumed by the one described in 20.1. If not, however, for more complex filtering logic, this technique will have its unique advantage in early and deep filtering.

22.3 The "other" Filters

The "other" filters, namely "non-pushdownable" filters, can be fully made as a custom filter. It will make the filtering logic earlier and deeper in the processing stack. Conceivably all single-table partition-specific filtering predicate can be made as a custom filter from the expression tree. That is, after setting the proper scan range and the possible dimensional skip, the original partition-specific filtering can be made as a custom filter. It should consume the technique as described in 20.2 as well as value filters.

23. Limitations

Currently, there is no support of versioned/timestamp-based queries on HBase, namely the queries always return the latest committed HBase data. No secure HBase support is in schedule either.

Another limitation is that the columns to be used in row keys have to be of primitive types or Strings: no complex data types can be used as part of the key.

24. Related Work

JIRA Spark-1127 is an attempt to address the issue of use of HBase as a data sink to Spark/SparkSQL.

This proposal is targeting more on i) tighter integration with SchameRDD, ii) online analysis capabilities, and iii) a unified interface for data analysts and admins of HBase data. For these purposes, the design features less dependence upon HBase's Map/Reduce interface, which, while provides great development convenience, incurs higher latency and lack of capabilities/flexibilities of utilization of some advanced HBase features such as coprocessors.

Apache Phoenix is another purpose-built SQL engine on top of HBase data. It is mainly driven by Salesforce.

25. Development Phases

For the complexity of the project, and the need to heed for community/user feedbacks, it is advisable that the development is phased. The first phase will see the DDL, DML, queries with filter and projection pushdowns and partition pruning. Future phases will see addition of custom filter support, which could be sub-phased for its own complexity, support of HTablePool, mixed-in of HBase shell, support of new HBase features like namespace and native data types, CTAS(create table as select), …, and others as demands arise.

More advanced optimizations for aggregation and join will come in later versions as well.

In the first release the physical key ordering is ascending only. The implication is that future sort-based optimization has to take this factor into consideration if and before the support of the descending ordering is in place.

26. Supported Spark Releases

As of July 2015, Spark 1.4 release will be supported. The plan is to support the latest Spark releases as soon as available.

27. Future Work

In both DDL and DML, the “IF (NOT) EXISTS” adjective will be supported in future releases.

Update of DML will be supported.

Composite join can be of immediate interests to the community as an effective way of joining two big tables of organized data.

BroadcastNestedLoopJoin might be pushed down to coprocessor.

In general, organized data stores present rich optimization opportunities for SQL engines. These opportunities will be explored in future.

HBase name space support will be added in the future and not in the first release.

Nowadays there exists a trend in the big data field in general, and in the HBase field in particular, to support transactional capabilities in SQL stores. This direction will certainly be worth of attention.

Some SQL features, such as Bloom Filter, have already had good support in HBase so the adoption cost might be minimal.

HBase 1.0 will be supported as well.

28. FAQs

1. Q: Is there any plan to support multiple scan threads per query per region?
   A: Multi scan threads could boost scan parallelism. The similar effect could be achieved through smaller regions. A possible down side of parallel scan is that it could result random I/O. Nevertheless it could be added readily as required.
2. Q: Are there other possible, yet not realized in the current release, optimization techniques on organized data sets like those on HBase?
   A: Yes. Actually there could be a bunch of such opportunities. We will, again, adopt them in a phased approach and would like to hear demands and feedbacks from the users.