Spark’s HBase API Design Doc

# Goals

To offer an easy to use API for interaction between Spark functionality and HBase that will support common use cases easily but also allow for very complex advance operations.

This API should manage the setting up and caching of one or more HConnections to HBase on the worker in the most optimal way for speed and reuse.

# Components

This solution is made up of two major components the HBaseContext and HConnectionStaticCache.

* **HBaseContext**: This will server as a façade in a similar way that the Graph class is used in the GraphX project. This is the only class in this design that the users will be interacting with. It will take a sparkContext and a HBaseConfiguration.
* **HConnectionStaticCache**: This scala object will serve as a static holding place for HConnections on a worker. It will handle the life cycle of a HConnection to best suit for reuse and resource management.

The next two following sections will dig deeper into these two component.

# HBaseContext

The HBaseContext is the soul interface the developer will have between Spark and HBase so to explain it we will go function by function.

Note that we will be talking about functions that will go into the first PR along with functions that will go into subsequence PRs. The reason for this is to help the reader of this document to understand the full hopes for this API but also to help us select the correct naming convention because it would be undesirable to change function names once they are released.

## Constructors

The following are the two constructors:

|  |
| --- |
| **HBaseContext(**@transient sc**:** **SparkContext,** broadcastedConf**:** **Broadcast[SerializableWritable[Configuration]])** |
| **def** **this(**@transient sc**:** **SparkContext,** @transient config**:** **Configuration)** |

In general the constructors take in a SparkContext and a Configuration object. The SparkContext is used in the distributedScan function to create a RDD from the HBase InputFormat. The configuration object is used in all interactions with HBase.

Note that the SparkContext and the Configuration objects are transient because we don’t want to send ether to the worker nodes. First there is no need for the SparkContext on the worker JVMs. Second we are using Spark’s broadcast functionality to only send the Configuration information to the worker once instead of sending for every partition on the worker.

# Functions

The functions of HBaseContext are blocked into the following groups of functions

* Common Functions
  + Bulk Put funcitons
  + Bulk Mutation functions
  + Bulk Get functions
  + Distributed Scan function
* Advanced Functions
  + ForeachPartition
  + MapPartition

Now there are a couple things to point out before we start going into the details of the different functions.

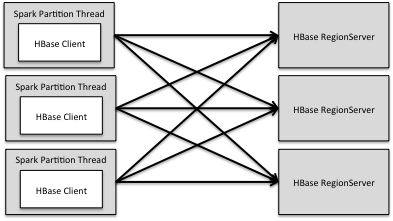
1. Some function blocks have “functions” not “function”. These are the Puts, Mutation, and Gets functions. The reason for this “s” is because there are different design patterns for performing these operations with HBase and we should to support the most common of these design patterns.
2. Note that there is a common and advance block. All the common functions (minus the distributed scan) are built on the advanced functions.
3. Note also there is a set of put functions and a mutation function. In HBase a Mutation is a put, increment, or delete. So why make a different set of functions for put from mutation? Well in the HTable implementation there is a option called autoFlush which will flush puts based on the number of bytes being cached on the client. This autoFlush option doesn’t apply to deletes and increments. Instead to do batching of deletes and increments we need to select a number to be batched together. So in short we have both Put and Mutation because the different interfaces in HTable allow for different methods of batching events.

### Bulk Put/Mutation Functionality

HBaseContext will support at least three Put/Mutation functions. All adding records, columns, and values to HBase but in very different ways. Each one is ideal for a different use case. Lets dig into the details in the following three sub sections.

### bulk/bulkMutation

The bulkPut/Mutation Put is the simplest of all the put/mutation patterns but also the worst and system utilization of resources and performance. But it is ideal for small put/mutation loads that may be common in some Spark Streaming use cases. The diagram below shows how the interaction between the Spark Partitions will interact with different HBase RegionServers while using bulkPut/Mutation method.



Note that all Spark Partitions are talking to all the RegionServers in the HBase cluster that contain regions of the given table. At a low level this means that the HBase client on each partition is using multiple threads and sending out more smaller batches of puts/mutations to a Region Server like the diagram blow shows.

Now smaller batches is not a good thing because of the way the WAL works on HBase. HBase needs to sync a batch of data to disk in order to return the put/mutation request. So the smaller the batches the more syncs for the same data volume.

The large the HBase Cluster and the more RegionServers a table is spanned across the worst this problem becomes. Just think about having 100 Region Servers. The bulkPut/Mutation solution would have ever put/mutation batch send out 100 RPC calls and then there will roughly be 100x the number of WAL syncs to disk.

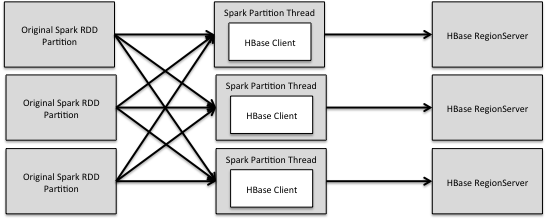
To address this waste we would have to partition the data before it gets sent to HBase. The partition would need to have knowledge of the Region splits that way every partition would only be focusing on a single Region. But for now lets wrap up with a summery of the bulkPut/Mutation option first

**Summery of BulkPut/Mutation**

* **Complexity**: Simple
* **Ideal use case**: Fast small batches
* **Anti-Pattern**: Not good for tables that span a lot of RegionServers
* **PR Release**: Initial release

### bulkPartitionedMutation/bulkPartitionedPut

The bulkPartitionedPut/Mutation is for the most part the same as the bulkPut,/Mutation except the data is partitioned by the key before going to HBase. This will result in each Spark partition is putting/mutating to a single Region. Which will mean larger batches and a fixed number of RPC threads for a give partition. So our load diagram will now change the to following.



So looking at the diagram we can see that we are trading off doubling the network for increased optimization to the batch sizes to the RegionServer.

It can be imagined conditions where this will be less efficient then the basic bulkPut/Mustation but because most of the time the additional network will be out weighed by the reducing in sync to the HBase WAL. Remember a RegionServers WAL is locked to a single drive at a time and they need to be replicated three times across the cluster so making sense to help make it write as efficient as possible.

**Summery of BulkPartitionedPut/Mutation**

* **Complexity**: Involves HBase integrated partition logic
* **Ideal use case**: Larger batches with table that span many Region Servers
* **Anti-Pattern**: Not the best for huge bulk loads and may be over kill for simple smaller bulk puts/mutations.
* **PR Release**: Future release

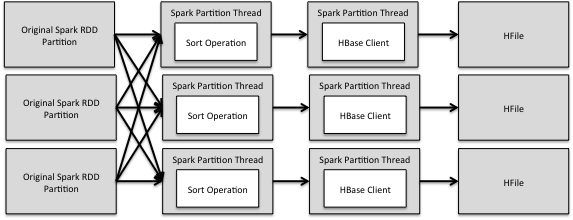
### bulkLoad

Now up until now both bulk solutions for getting data into HBase have been centered on puts and mutations done through a HTable object. Puts/Mutations are great but they have a major flew. If the WAL is left on a Put/Mutation must pass through a RegionServers wall. This means a couple things.

* All puts/mutations for a Region Server must go through a single drive
* All puts/mutations must be replicated too times: for the WAL and the final HFile drop
* HFile sizes are limited to the flush sized defined in the Region Server (for those new to HBase larger and less HFiles are better then smaller ad more HFile in a Region Server)

So we can solve all these problems by avoiding the WAL by writing straight to HFiles. This functionality already exist in HBase as a MR job, but it is purposed that we also implement this with Spark which should be faster then it’s MR brother.

The solution is pretty simple, we just need to partition the data by the region splits then sort with in a given partition then do a foreachPartition to write out the given HFile. This process will look like the following:



Once the HFiles are written to HDFS there is a simple HBase command to load these HFile into a give table.

At scale a bulkLoad can be more then 2x time faster then a bulkPartitionedPut/Mutation so this option should defiantly be included as an option in our HBaseContext.

**Summery of BulkPartitionedPut/Mutation**

* **Complexity**: Involves HBase integrated partition logic and HFile writing logic
* **Ideal use case**: Largest of Loads
* **Anti-Pattern**: Not good for small batches because it may make smaller HFiles then the memstor flush size
* **PR Release**: Future release

## Creating RDDs From HBase Data Functionality

Unlike all the putting and mutating we are no longer putting data into HBase. In the following functions we are going to take data from HBase and create RDDs in Spark.

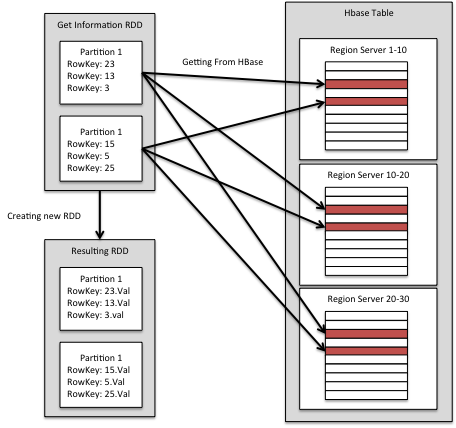
Like the put/mutation functions there are three ways we can get data from HBase:

* BulkGet
* BulkPartitionedSortedGet
* Scan

Just like the put/mutation the different functions are different for different use cases, so lets dig into the details.

### BulkGet

The BulkGet function will take a RDD from Spark and generate a list of Get operations to request information from HBase to generate a new RDD. The diagram below will hope to define what this process will look like.



So in short we use the initial RDD to look up data from HBase and use that looked up data to create a new RDD.

So just like the basic bulkPut/Mutation the bulk Get is good for small quick fetching of data from Hbase in a parallel way. A good use case would be during a Spark Streaming process to do short burst of gets that need to get back fast.

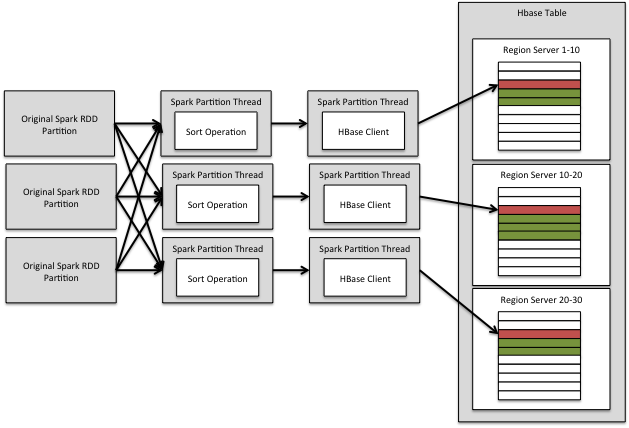
The reasons this is not good for massive amounts of Gets is the Gets are not ordered and not partitioned by region, so the gets are scattered across all the RegionServers and the likely hood of none-blockcache hits is very high.

**Summery of BulkGets**

* **Complexity**: Simple
* **Ideal use case**: small set of Gets most likely in a Spark Streaming use case
* **Anti-Pattern**: Not good for large gets, scans, or co-located gets
* **PR Release**: Initial release

### BulkPartitionedSortedGet

So like the bulkLoad we are prepping the data so that we can get better performance out of HBase. In this case we are partitioning the data so that each partition is only talking to the minimum number of region servers as possible at a time. Plus the sorting will allow gets of co-located rows to avoid seeks on disk and use the block cache instead.



The first thing to note about the diagram each partition is talking to as few Regions as possible and the second thing to note is the green records are records we are able to get through the block cache instead of a disk seek.

**Summery of BulkPartitionSortedGet**

* **Complexity**: Involves HBase integrated partition logic
* **Ideal use case**: When you need to do lots of Gets but the reaches are scattered across that table in that a scan is over kill.
* **Anti-Pattern**: Good for large scale when getting a sub set of records scattered through out a table
* **PR Release**: Future release

### DistributedScan

So if all the records you need to get are next to each other with in range or if you want to get the whole table then getting is not the right answer. Instead you want to do a straight distributed scan.

To implement this HBaseContext defers the scanning logic to the HBaseInputFormat and produces a RDD.

**Summery of DistributedScan**

* **Complexity**: Simple
* **Ideal use case**: Needing to read long stretches of data from HBase
* **Anti-Pattern**: Good for small to super large datasets.
* **PR Release**: Initial release

### Advanced Functions

So sometimes the use case is complex and the developer may want to do gets and mutations in the same process. So these operations we are exposing the foreachPartition and mapPartition functions.

These functions take place after the HConnection is made and end before the HConnection is returned to the cache. This allows the developer full freedom to do any operation on as many tables as they wish.

**Summery of Advanced Functions**

* **Complexity**: Simple
* **Ideal use case**: Advanced developer use cases
* **Anti-Pattern**: All functions by distributed scan are based on these functions so the biggest issue here is misuse.
* **PR Release**: Initial release

# HConnectionStaticCache

Unlike the HBaseContext the HConnectionStatic Cache is something that a developer should never interact with. Also it should only be present on the Spark Workers and not on the client JVM. It’s job is to hold HConnection to avoid creating more then need. This is even move important with it comes to Spark Streaming.

How HConnectionStaticCahce works is pretty Simple it has a static hash map that has a key that is the instance id from the configuration object “hbase.client.instance.id” or HConstants.HBASE\_CLIENT\_INSTANCE\_ID. This map will allow us to connect to more then one HBase cluster in a give Worker or to even have different client connection attributes to the same HBase cluster in a single Worker.

The value of the hashMap is three parts.

* First it is the HConnection that will be reused
* Second is a counter of how many threads that are using the HConnection
* Third is the time when the number of threads used was set back to zero.

The HConnectionStaticCache also has a Timer thread that will wake up every 120 seconds. When it wakes up it will look through all the HConnections in the HashMap. It if finds any HConnection with a usage counter of Zero and a last unused more then 60 seconds ago then it will remove and close the HConnection.

The time gap between unused and remove/close is for Spark Stream or back to back jobs that could be using the same connection information.