# Apache HBase

Wednesday Feb 01, 2012

Coprocessor Introduction

**Authors: Trend Micro Hadoop Group: Mingjie Lai, Eugene Koontz, Andrew Purtell**

(The original version of the blog was posted at http://hbaseblog.com/2010/11/30/hbase-coprocessors/ in late 2010, however the site is no longer available. Since we decided to move all blog posts to the official Apache blog, here I just recover the original post and make quite some modifications to reflect the latest coprocessor changes in HBase 0.92.)

HBase has very effective MapReduce integration for distributed computation over data stored within its tables, but in many cases – for example simple additive or aggregating operations like summing, counting, and the like – pushing the computation up to the server where it can operate on the data directly without communication overheads can give a dramatic performance improvement over HBase's already good scanning performance.

Also, before 0.92, it was not possible to extend HBase with custom functionality except by extending the base classes. Due to Java's lack of multiple inheritance this required extension plus base code to be refactored into a single class providing the full implementation, which quickly becomes brittle when considering multiple extensions. Who inherits from whom? Coprocessors allow a much more flexible mixin extension model.

In this article I will introduce the new Coprocessors feature of HBase, a framework for both flexible and generic extension, and of distributed computation directly within the HBase server processes. I will talk about what it is, how it works, and how to develop coprocessor extensions.

The idea of HBase Coprocessors was inspired by Google's BigTable coprocessors. Jeff Dean gave a talk at LADIS '09 (http://www.scribd.com/doc/21631448/Dean-Keynote-Ladis2009, page 66-67), and mentioned some basic ideas of coprocessors, which Google developed to bring computing parallelism to BigTable. They have the following characteristics:

- Arbitrary code can run at each tablet in table server
- High-level call interface for clients
    - Calls are addressed to rows or ranges of rows and the coprocessor client library resolves them to actual locations;
    - Calls across multiple rows are automatically split into multiple parallelized RPC
- Provides a very flexible model for building distributed services
- Automatic scaling, load balancing, request routing for applications

Back to HBase, we definitely want to support efficient computational parallelism as well, beyond what Hadoop MapReduce can provide. In addition, exciting new features can be built on top of it, for example secondary indexing, complex filtering (push down predicates), and access control. HBase coprocessors are inspired by BigTable coprocessors but are divergent in implementation detail. What we have built is a framework that provides a library and runtime environment for executing user code within the HBase region server and master processes. Google coprocessors in contrast run colocated with the tablet server but outside of its address space. (HBase is also considering an option for deployment of coprocessor code external to the server process. See https://issues.apache.org/jira/browse/HBASE-4047 .)

Coprocessors can be loaded globally on all tables and regions hosted by the region server, these are known as system coprocessors; or the administrator can specify which coprocessors should be loaded on all regions for a table on a per-table basis, these are known as table coprocessors.

In order to support sufficient flexibility for potential coprocessor behaviors, two different aspects of extension are provided by the framework. One is the observer, which are like triggers in conventional databases, and the other is the endpoint, dynamic RPC endpoints that resemble stored procedures.

## Observers

The idea behind observers is that we can insert user code by overriding upcall methods provided by the coprocessor framework. The callback functions are executed from core HBase code when certain events occur. The coprocessor framework handles all of the details of invoking callbacks during various base HBase activities; the coprocessor need only insert the desired additional or alternate functionality.

Currently in HBase 0.92, we have three observer interfaces provided:

- RegionObserver: Provides hooks for data manipulation events, Get, Put, Delete, Scan, and so on. There is an instance of a RegionObserver coprocessor for every table region and the scope of the observations they can make is constrained to that region.
- WALObserver: Provides hooks for write-ahead log (WAL) related operations. This is a way to observe or intercept WAL writing and reconstruction events. A WALObserver runs in the context of WAL processing. There is one such context per region server.
- MasterObserver: Provides hooks for DDL-type operation, i.e., create, delete, modify table, etc. The MasterObserver runs within the context of the HBase master.

More than one observer can be loaded at one place -- region, master, or WAL -- to extend functionality. They are chained to execute sequentially by order of assigned priorities. There is nothing preventing a coprocessor implementor from communicating internally between the contexts of his or her installed observers, providing comprehensive coverage of HBase functions.

A coprocessor at a higher priority may preempt action by those at lower priority by throwing an IOException (or a subclass of this). We will use this preemptive functionality below in an example of an Access Control coprocessor.

As we mentioned above, various events cause observer methods to be invoked on loaded observers. The set of events and method signatures are presented in the HBase API, beginning with HBase version 0.92. Please be aware that the API could be changed in future, due to the HBase Client API changes, or possibly other reasons. We've tried to stablize the API before 0.92 release but there is no guarantee.)

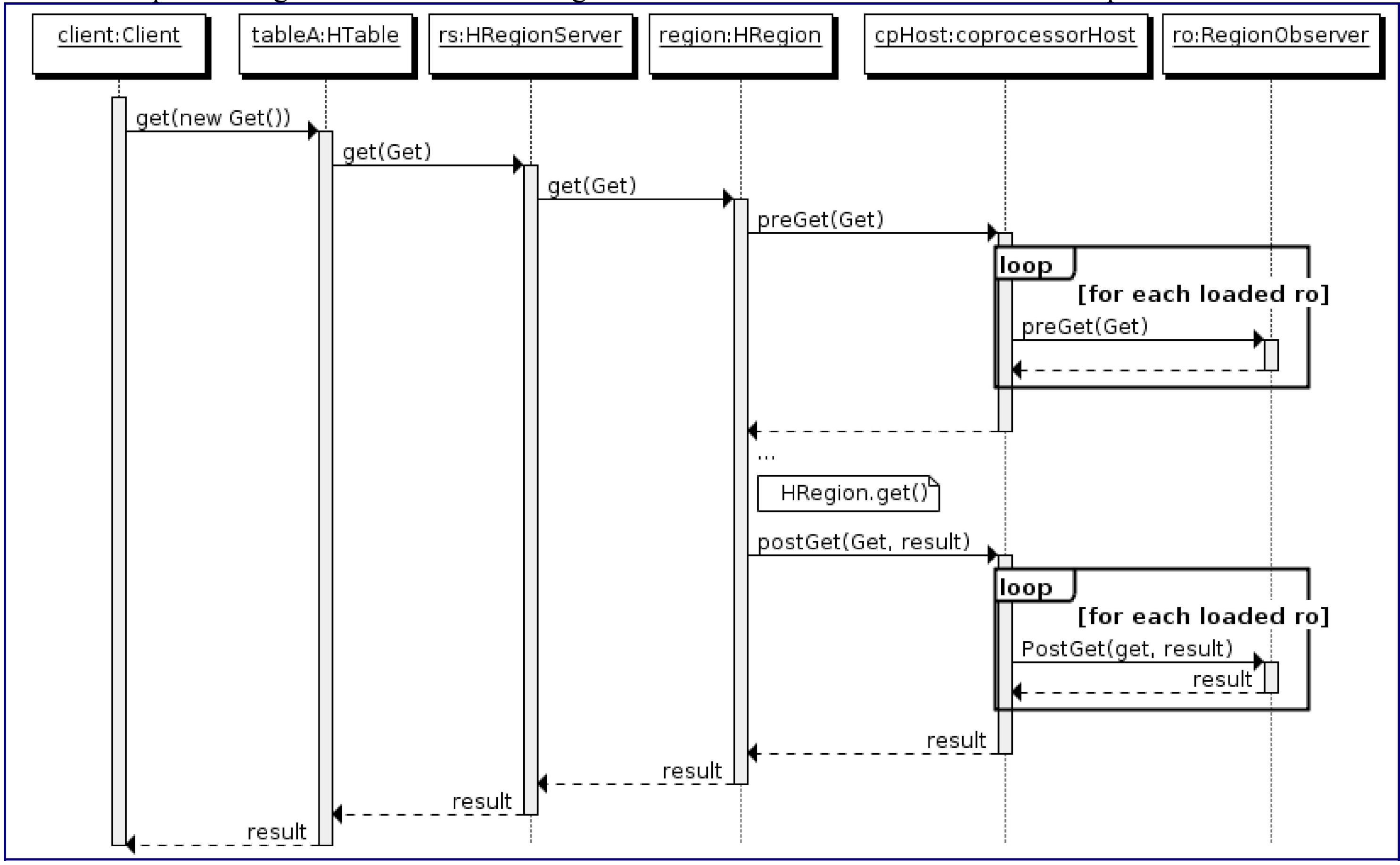
The RegionObserver interface provides callbacks for:

- preOpen, postOpen: Called before and after the region is reported as online to the master.
- preFlush, postFlush: Called before and after the memstore is flushed into a new store file.
- preGet, postGet: Called before and after a client makes a Get request.
- preExists, postExists: Called before and after the client tests for existence using a Get.
- prePut and postPut: Called before and after the client stores a value.
- preDelete and postDelete: Called before and after the client deletes a value.
- etc.

Please refer to HBase 0.92 javadoc to get the whole list of declared methods.

We provide a convenient abstract class BaseRegionObserver, which implements all RegionObserver methods with default behaviors, so you can focus on what events you have interest in, without having to be concerned about process upcalls for all of them.

Here is a sequence diagram that shows how RegionObservers works with other HBase components:



Below is an example of an extension that hooks into HBase function using the RegionObserver interface. It is a demonstration of the addition of simple access control. This coprocessor checks user information for a given client request by injecting code at certain RegionObserver preXXX hooks. If the user is not allowed to access the resource, an AccessDeniedException will be thrown. As we mentioned above, exceptions by high-priority coprocessors (such as an access control coprocessor) can be used to preempt further action. In this case, the AccessDeniedException means that the client's request will not be processed and the client will receive the exception as indication of what happened.

```
package org.apache.hadoop.hbase.coprocessor;

import java.util.List;
import org.apache.hadoop.hbase.KeyValue;
import org.apache.hadoop.hbase.client.Get;

// Sample access-control coprocessor. It utilizes RegionObserver
// and intercept preXXX() method to check user privilege for the given table
// and column family.
public class AccessControlCoprocessor extends BaseRegionObserver {
  @Override
  public void preGet(final ObserverContext<RegionCoprocessorEnvironment> c,
final Get get, final List<KeyValue> result) throws IOException
      throws IOException {

    // check permissions..
    if (!permissionGranted())  {
        throw new AccessDeniedException("User is not allowed to access.");
    }
  }

  // override prePut(), preDelete(), etc.
}
```

The MasterObserver interface provides upcalls for:

- preCreateTable/postCreateTable: Called before and after the region is reported as online to the master.
- preDeleteTable/postDeleteTable
- etc.

WALObserver provides upcalls for:

- preWALWrite/postWALWrite: called before and after a WALEdit written to WAL.

Please refer to the HBase javadoc for the whole list of observer interface declarations.

## Endpoint

As mentioned previously, observers can be thought of like database triggers. Endpoints, on the other hand, are more powerful, resembling stored procedures. One can invoke an endpoint at any time from the client. The endpoint implementation will then be executed remotely at the target region or regions, and results from those executions will be returned to the client.

Endpoint is an interface for dynamic RPC extension. The endpoint implementation is installed on the server side and can then be invoked with HBase RPC. The client library provides convenience methods for invoking such dynamic interfaces.

Also as mentioned above, there is nothing stopping the implementation of an endpoint from communicating with any observer implementation. With these extension surfaces combined you can add whole new features to HBase without modifying or recompiling HBase itself. This can be very powerful.

In order to build and use your own endpoint, you need to:

- Have a new protocol interface which extends CoprocessorProtocol.
- Implement the Endpoint interface. The implementation will be loaded into and executed from the region context.
- Extend the abstract class BaseEndpointCoprocessor. This convenience class hides some internal details that the implementer need not necessary be concerned about, such as coprocessor framework class loading.
- On the client side, the Endpoint can be invoked by two new HBase client APIs:
    - Executing against a single region:

      ```
      HTableInterface.coprocessorProxy(Class<T> protocol, byte[] row)
      ```

    - Executing over a range of regions

      ```
      HTableInterface.coprocessorExec(Class<T> protocol, byte[] startKey, byte[]
      endKey, Batch.Call<T,R> callable)
      ```

Here is an example that shows how an endpoint works.

In this example, the endpoint will scan a given column in a region and aggregate values, expected to be serialized Long values, then return the result to client. The client collects the partial aggregates returned from remote endpoint invocation over individual regions, and sums the results to get the final answer over the full table. Note that the HBase client has the responsibility for dispatching parallel endpoint invocations to the target regions, and for collecting the returned results to present to the application code. This is like a lightweight MapReduce job: The "map" is the endpoint execution performed in the region server on every target region, and the "reduce" is the final aggregation at the client. Meanwhile, the coprocessor framework on the server side and in the client library is like the MapReduce framework, moving tedious distributed systems programming details behind a clean API, so the programmer can focus on the application.

Note also that HBase, and all of Hadoop, currently requires Java 6, which has a verbose syntax for anonymous classes. As HBase (and Hadoop) evolves with the introduction of Java 7 language features, we expect the verbosity of client endpoint code can be reduced substantially.

```
// A sample protocol for performing aggregation at regions.
public static interface ColumnAggregationProtocol
extends CoprocessorProtocol {
  // Perform aggregation for a given column at the region. The aggregation
  // will include all the rows inside the region. It can be extended to
  // allow passing start and end rows for a fine-grained aggregation.
  public long sum(byte[] family, byte[] qualifier) throwsIOException;
}

// Aggregation implementation at a region.
public static class ColumnAggregationEndpoint extends BaseEndpointCoprocessor
implements ColumnAggregationProtocol {
  @Override
  public long sum(byte[] family, byte[] qualifier)
  throws IOException {
    // aggregate at each region
    Scan scan = new Scan();
    scan.addColumn(family, qualifier);
    long sumResult = 0;
    InternalScanner scanner = getEnvironment().getRegion().getScanner(scan);
    try {
      List<KeyValue> curVals = new ArrayList<KeyValue>();
      boolean hasMore = false;
      do {
    curVals.clear();
    hasMore = scanner.next(curVals);
    KeyValue kv = curVals.get(0);
    sumResult += Bytes.toLong(kv.getValue());
      } while (hasMore);
    } finally {
        scanner.close();
    }
    return sumResult;
  }
}
```

Client invocations are performed through new methods on HTableInterface:

```
public <T extends CoprocessorProtocol> T coprocessorProxy(Class<T> protocol, Row row);

public <T extends CoprocessorProtocol, R> void coprocessorExec(
    Class<T> protocol, List<? extends Row> rows,
    BatchCall<T,R> callable, BatchCallback<R> callback);

public <T extends CoprocessorProtocol, R> voidcoprocessorExec(
    Class<T> protocol, RowRange range,
    BatchCall<T,R> callable, BatchCallback<R> callback);
```

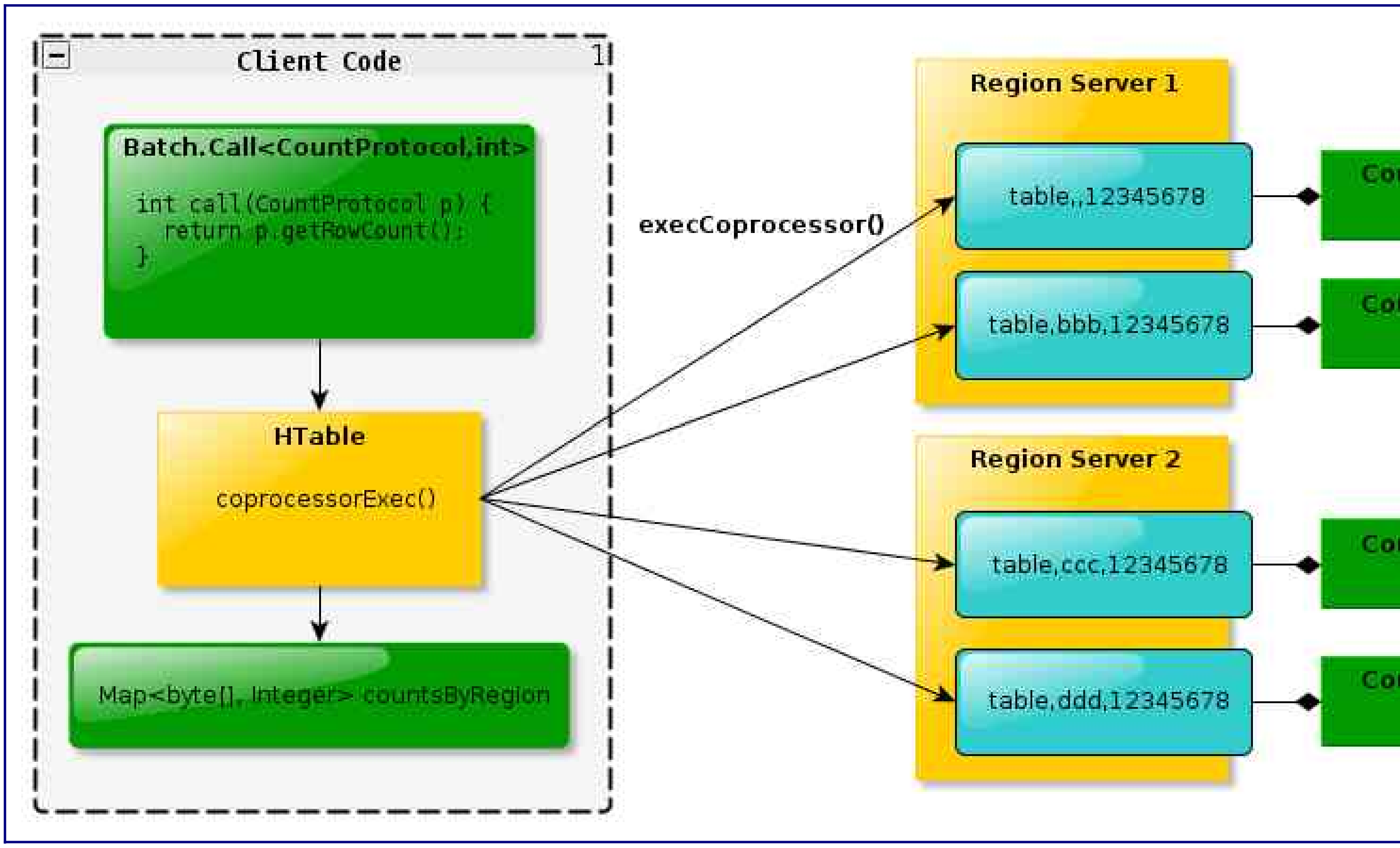
Here is the client side example of calling ColumnAggregationEndpoint:

```
HTableInterface table = new HTable(util.getConfiguration(), TEST_TABLE);
Scan scan;
Map<byte[], Long> results;

// scan: for all regions
scan = new Scan();
results = table.coprocessorExec(ColumnAggregationProtocol.class, scan,
    new BatchCall<ColumnAggregationProtocol,Long>() {
      public Integer call(ColumnAggregationProtocol instance)throws IOException{
        return instance.sum(TEST_FAMILY, TEST_QUALIFIER);
      }
    });
long sumResult = 0;
long expectedResult = 0;
for (Map.Entry<byte[], Long> e : results.entrySet()) {
  sumResult += e.getValue();
}
```

The above example is actually a simplified version of HBASE-1512. You can refer to that JIRA or the HBase source code of org.apache.hadoop.hbase.coprocessor.AggregateImplementation for more detail.

Below is a visualization of dynamic RPC invocation for this example. The application code client side performs a batch call. This initiates parallel RPC invocations of the registered dynamic protocol on every target table region. The results of those invocations are returned as they become available. The client library manages this parallel communication on behalf of the application, messy details such as dealing with retries and errors, until all results are returned (or in the event of an unrecoverable error). Then the client library rolls up the responses into a Map and hands it over to the application. If an unrecoverable error occurs, then an exception will be thrown for the application code to catch and take action.



## Coprocessor Management

After you have a good understanding of how coprocessors work in HBase, you can start to build your own experimental coprocessors, deploy them to your HBase cluster, and observe the new behaviors.

### Build Your Own Coprocessor

We now assume you have your coprocessor code ready, compiled and packaged as a jar file. You will see how coprocessor framework can be configured to load the coprocessor in the following sections.

(We should have a template coprocessor that helps users quickly start to develop. Currently there are some built-in coprocessors that can serve as examples and a starting point for implementation of a new coprocessor. However they are scattered over the code base. As discussed in HBASE-5273, there will be some coprocessors samples provided under src/example/coprocessor of the HBase source code. )

### Coprocessor Deployment

Currently we provide two options for deploying coprocessor extensions: load from configuration, which happens when the master or region servers start up; or load from table attribute, dynamic loading when the table is (re)opened. Because most users will set table attributes by way of the ‘alter’ command of the HBase shell, let’s call this load from shell.

#### Load from Configuration

When a region is opened, the framework tries to read coprocessor class names supplied as the configuration entries:

- hbase.coprocessor.region.classes: for RegionObservers and Endpoints
- hbase.coprocessor.master.classes: for MasterObservers
- hbase.coprocessor.wal.classes: for WALObservers

Hers is an example of the hbase-site.xml where one RegionObserver is configured for all the HBase tables:

```
<property>
    <name>hbase.coprocessor.region.classes</name>
    <value>org.apache.hadoop.hbase.coprocessor.AggregateImplementation</value>
 </property>
```

If there are multiple classes specified for loading, the class names must be comma separated. Then, the framework will try to load all the configured classes using the default class loader. This means the jar file must reside on the server side HBase classpath.

If loaded in this manner, the coprocessors will be active on all regions of all tables. This is the system coprocessor as earlier introduced. The first listed coprocessors will be assigned the priority Coprocessor.Priority.SYSTEM. Each subsequent coprocessor in the list will have its priority value incremented by one (which reduces its priority, priorities have the natural sort order of Integers).

We have not really discussed priority, but it should be reasonably clear how the priority given to a coprocessor affects how it integrates with other coprocessors. When calling out to registered observers, the framework executes their callbacks methods in the sorted order of their priority. Ties are broken arbitrarily.

**Load from Shell**

Coprocessors can also be configured to load on a per table basis, via a shell command ``alter'' + ``table_att''.

```
hbase(main):005:0>  alter 't1', METHOD => 'table_att',
  'coprocessor'=>'hdfs:///foo.jar|com.foo.FooRegionObserver|1001|arg1=1,arg2=2'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.0730 seconds

hbase(main):006:0> describe 't1'
DESCRIPTION                                                        ENABLED
 {NAME => 't1', coprocessor$1 => 'hdfs:///foo.jar|com.foo.FooRegio false
 nObserver|1001|arg1=1,arg2=2', FAMILIES => [{NAME => 'c1', DATA_B
 LOCK_ENCODING => 'NONE', BLOOMFILTER => 'NONE', REPLICATION_SCOPE
  => '0', VERSIONS => '3', COMPRESSION => 'NONE', MIN_VERSIONS =>
 '0', TTL => '2147483647', KEEP_DELETED_CELLS => 'false', BLOCKSIZ
 E => '65536', IN_MEMORY => 'false', ENCODE_ON_DISK => 'true', BLO
 CKCACHE => 'true'}, {NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE',
  BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSIONS => '3'
 , COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => '2147483647'
 , KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY
 => 'false', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0190 seconds
```

The coprocessor framework will try to read the class information from the coprocessor table attribute value. The value contains four pieces of information which are separated by ``|”:

- File path: The jar file containing the coprocessor implementation must located somewhere where all region servers can read it. The file could be copied somewhere onto the local disk of all region servers, but we recommended storing the file into HDFS instead. If no file path is given, the framework will attempt to load the class from the server classpath using the default class loader.
- Class name: The full class name of the coprocessor.
- Priority: An integer. The framework will determine the execution sequence of all configured observers registered at the same hook using priorities. This field can be left blank. In that case the framework will assign a default priority value.
- Arguments: This field is passed to the coprocessor implementation.

You can also remove a loaded coprocessor at shell, by ``alter'' + ``table_att_unset'' command:

```
hbase(main):007:0> alter 't1', METHOD => 'table_att_unset',
hbase(main):008:0*   NAME => 'coprocessor$1'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.1130 seconds

hbase(main):009:0> describe 't1'
DESCRIPTION                                                        ENABLED
 {NAME => 't1', FAMILIES => [{NAME => 'c1', DATA_BLOCK_ENCODING => false
  'NONE', BLOOMFILTER => 'NONE', REPLICATION_SCOPE => '0', VERSION
 S => '3', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => '214
 7483647', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN
 _MEMORY => 'false', ENCODE_ON_DISK => 'true', BLOCKCACHE => 'true
 '}, {NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER =>
 'NONE', REPLICATION_SCOPE => '0', VERSIONS => '3', COMPRESSION =>
  'NONE', MIN_VERSIONS => '0', TTL => '2147483647', KEEP_DELETED_C
 ELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', ENCO
 DE_ON_DISK => 'true', BLOCKCACHE => 'true'}]}
1 row(s) in 0.0180 seconds
```

(Note: this is the current behavior in 0.92. There has been some discussion of refactoring how table attributes are handled for coprocessors and other features using the shell’s ‘alter’ command. See HBASE-4879 for more detail.)

In this section, we discussed how to tell HBase which coprocessors are expected to be loaded, but there is no guarantee that the framework will load them successfully. For example, the shell command neither guarantees a jar file exists at a particular location nor verifies if the given class is actually contained in the jar file.

### HBase Shell Coprocessor Status

After a coprocessor has been configured, you also need to check the coprocessor status using the shell or master and region server web UIs to determine if the coprocessor has been loaded successfully.

Shell command:

```
hbase(main):018:0>  alter 't1', METHOD => 'table_att',
  'coprocessor'=>'|org.apache.hadoop.hbase.coprocessor.AggregateImplementation|1001|
arg1=1,arg2=2'
Updating all regions with the new schema...
1/1 regions updated.
Done.
0 row(s) in 1.1060 seconds

hbase(main):019:0> enable 't1'
0 row(s) in 2.0620 seconds

hbase(main):020:0> status 'detailed'
version 0.92-tm-6
0 regionsInTransition
master coprocessors: []
1 live servers
    localhost:52761 1328082515520
        requestsPerSecond=3, numberOfOnlineRegions=3, usedHeapMB=32, maxHeapMB=995
        -ROOT-,,0
            numberOfStores=1, numberOfStorefiles=1, storefileUncompressedSizeMB=0,
storefileSizeMB=0, memstoreSizeMB=0,
storefileIndexSizeMB=0, readRequestsCount=54, writeRequestsCount=1, rootIndexSizeKB=0,
totalStaticIndexSizeKB=0,
totalStaticBloomSizeKB=0, totalCompactingKVs=0, currentCompactedKVs=0,
compactionProgressPct=NaN, coprocessors=[]
        .META.,,1
            numberOfStores=1, numberOfStorefiles=0, storefileUncompressedSizeMB=0,
storefileSizeMB=0, memstoreSizeMB=0,
storefileIndexSizeMB=0, readRequestsCount=97, writeRequestsCount=4, rootIndexSizeKB=0,
totalStaticIndexSizeKB=0,
totalStaticBloomSizeKB=0, totalCompactingKVs=0, currentCompactedKVs=0,
compactionProgressPct=NaN, coprocessors=[]
        t1,,1328082575190.c0491168a27620ffe653ec6c04c9b4d1.
            numberOfStores=2, numberOfStorefiles=1, storefileUncompressedSizeMB=0,
storefileSizeMB=0, memstoreSizeMB=0,
storefileIndexSizeMB=0, readRequestsCount=0, writeRequestsCount=0, rootIndexSizeKB=0,
totalStaticIndexSizeKB=0,
totalStaticBloomSizeKB=0, totalCompactingKVs=0, currentCompactedKVs=0,
compactionProgressPct=NaN,
coprocessors=[AggregateImplementation]
0 dead servers
```

If you cannot find the coprocessor loaded, you need to check the server log files to discover the reason for its failure to load.

## Current Status

There are several JIRAs opened for coprocessor development. HBASE-2000 functioned as the umbrella for coprocessor development. HBASE-2001 covered coprocessor framework development. HBASE-2002 covered RPC extensions for endpoints. Code resulting from work on these issues was committed to HBase trunk in 2010, and are available beginning with the 0.92.0 release.

There are some new features developed on top of coprocessors:

- HBase access control (HBASE-3025, HBASE-3045): This is experimental basic access control for HBase, available in 0.92.0, but unlikely to be fully functional prior to 0.92.1.
- Column aggregates (HBASE-1512): Providing experimental support for SQL-like sum(), avg(), max(), min(), etc. functions over columns. Available in 0.92.0.
- Constraints (HBASE-4605): A mechanism for simple restrictions on the domain of a data

attribute. For more information read the Constraints package summary: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/constraint/package-summary.html . It’s in trunk, will be available in the 0.94 releases.

## Future Work

Coprocessors are a brand new feature available starting with the 0.92.0 release. While they open many doors, there is much room for improvement. More hooks and interfaces may be added over time to support more use cases, such as:

### Parallel Computation Framework

By way of endpoints, we have a new dynamic way to inject user code into the processing of actions on individual table regions, and with the corresponding client side support we can interrogate them all in parallel and return results to the client in a flexible manner. This is immediately useful for building batch data processing and aggregation on top of HBase. However you need to understand some internal HBase detail to develop such applications.

Various JIRAs have been opened that consider exposing additional framework for parallel computation that can provide a convenient but powerful higher level of abstraction. Options under consideration include MapReduce APIs similar to those provided by Hadoop; support for scriptlets, i.e. Ruby script fragments sent server side to perform work; or something akin or directly supporting the Cascading framework (http://cascading.org) on the server for working with data flows more abstractly.

However, as far as I know, none of them is under construction right now.

### External Coprocessor Host (HBASE-4047)

Where HBase coprocessors deviate substantially from the design of Google's BigTable coprocessors is we have reimagined them as a framework for internal extension. In contrast, BigTable coprocessors run as separate processes colocated with tablet servers. The essential trade off is between performance, flexibility and possibility; and the ability to control and enforce resource usage.

We are considering developing a coprocessor that is a generic host for another coprocessor. The host installs in-process into the master or region servers, but the user code will be loaded into a forked child process. An eventing model and umbilical protocol over a bidirectional pipe between the parent and child will provide the user code loaded into the child the same semantics as if it were loaded internally to the parent. However, we immediately isolate the user code from HBase internals, and eventually expect to use available resource management capabilities on the platform to further limit child resource consumption as desired by system administrators or the application designer.

### Code Weaving (HBASE-2058)

Right now there are no constraints on what actions a coprocessor can take. We do not protect against malicious actions or faults accidentally introduced by a coprocessor. As an alternative to external coprocessor hosts we could introduce code weaving and code transformation at coprocessor load time. We would weave in a configurable set of policies for constraining the actions a coprocessor can take. For example:

- Improve fault isolation and system integrity protections via various static checks
- Wrap heap allocations to enforce limits
- Monitor CPU time via instrumentation injected into method and loop headers

- Reject static or dynamic method invocations to APIs considered unsafe

**And More...**

Coprocessors framework provides possibilities to extend HBase. There are some more identified applications which can be built on top of coprocessors:

- HBase isolation and allocation (HBase-4120)
- Secondary indexing: http://wiki.apache.org/hadoop/Hbase/SecondaryIndexing
- Search in HBase (HBASE-3529)
- HBase table, region access statistic.
- and more ...

Posted at 08:12AM Feb 01, 2012 by mlai in General | Comments[3] |

Comments:

[Trackback] Scalability Scalability of HadoopBig Data and Hado

Posted by **Confluence: Pythia** on February 13, 2012 at 03:32 PM GMT+00:00 #

[Trackback] Scalability Scalability of HadoopBig Data and Hado

Posted by **Confluence: Pythia** on February 13, 2012 at 03:46 PM GMT+00:00 #

This has been very helpful. Thanks for the writeup.