User documentation for High Available Reads

This doc will be committed as a part of https://issues.apache.org/jira/browse/HBASE-10513.

Introduction

HBase, architecturally, always had the strong consistency guarantee from the start. All reads and writes are routed through a single region server, which guarantees that all writes happen in an order, and all reads are seeing the most recent committed data.

However, because of this single homing the reads to a single location, if the server becomes unavailable, the regions of the table that were hosted in the region server becomes unavailable for some time. There are three phases in the MTTR process - detection, assignment, and recovery. Of these, the detection is usually the longest and is presently in the order of 20-30 seconds depending on the zookeeper session timeout. During this time and before the recovery is complete, the clients would not be able to read the region data.

However, for some use cases, either the data may be read-only, or doing reads againsts some stale data is acceptable. With High Available Reads, HBase can be used for these kind of latency-sensitive use cases where the application can expect to have a time bound on the read completion.

For achieving high availability for reads, HBase provides a feature called “region replication”. In this model, for each region of a table, there will be multiple replicas that are opened in different region servers. By default, the region replication is set to 1, so only a single region replica is deployed and there will not be any changes from the previous model. If region replication is set to 2 or more, than the master will assign replicas of the regions of the table. The Load Balancer ensures that the region replicas are not co-hosted in the same region servers and also in the same rack (if possible).

All of the replicas for a single region will have a unique replica_id, starting from 0. The region replica having replica_id==0 is called the primary region, and the others “secondary regions” or secondaries. Only the primary can accept writes from the client, and the primary will always contain the latest changes. Since all writes still has to go through the primary region, the writes are not high-available (meaning they might block for some time if the region becomes unavailable).

The writes are asynchronously sent to the secondary region replicas using “Async WAL replication” feature. This works similarly to HBase’s multi-datacenter replication, but instead the data from a region is replicated to the secondary regions. Each secondary replica always receives and observes the writes in the same order that the primary region committed them. This ensures that the secondaries won’t diverge from the primary regions data, but since the log

replication is asnyc, the data might be stale in secondary regions. In some sense, this design can be thought as “in-cluster replication”, where instead of replicating to a different datacenter, the data goes to a secondary region where the client can read from. The data files are shared between the primary region and the other replicas, so that there is no extra storage overhead.

Timeline Consistency

With this feature, HBase introduces a `Consistency` definition, which can be provided per read operation (get or scan).

```
public enum Consistency {
  STRONG,
  TIMELINE
}
```

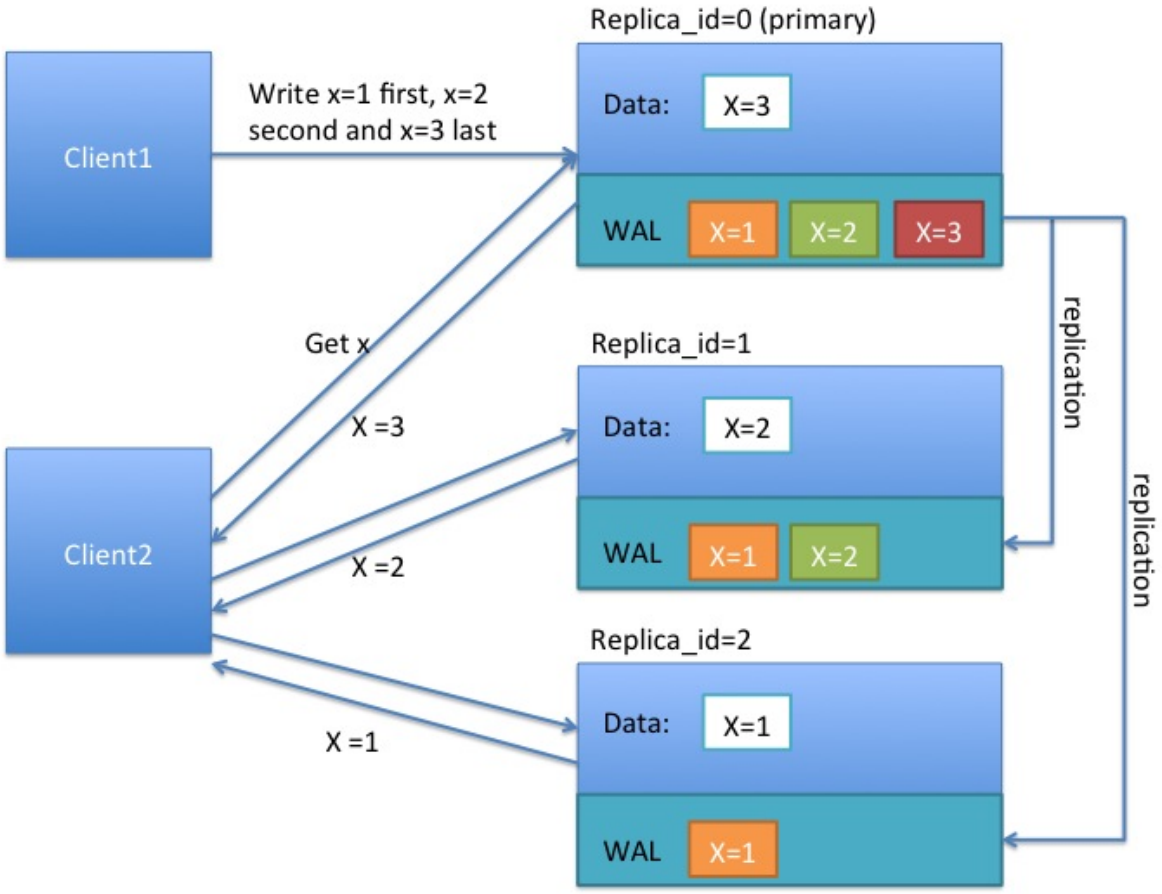
`Consistency.STRONG` is the default consistency model provided by HBase. In case the table has region replication = 1, or in a table with region replicas but the reads are done with this consistency, the read is always performed by the primary regions, so that there will not be any change from the previous behaviour, and the client always observes the latest data.

In case a read is performed with `Consistency.TIMELINE`, then the read RPC will be sent to the primary region server first. After a short interval (hbase.client.primaryCallTimeout.get, 10ms by default), parallel RPC for secondary region replicas will also be sent if the primary does not respond back. After this, the result is returned from whichever RPC is finished first. If the response came back from the primary region replica, we can always know that the data is latest. For this Result.isStale() API has been added to inspect the staleness. If the result is from a secondary region, then Result.isStale() will be set to true. The user can then inspect this field to possibly reason about the data.

In terms of semantics, TIMELINE consistency as implemented by HBase differs from pure eventual consistency in these respects:

- Single homed and ordered updates: Region replication or not, on the write side, there is still only 1 defined replica (primary) which can accept writes. This replica is responsible for ordering the edits and prevent conflicts. This guarantees that two different writes are not committed at the same time by different replicas and the data diverges. With this, there is no need to do read-repair or last-timestamp-wins kind of conflict resolution.
- The secondaries also apply the edits in order that the primary committed them. This way the secondaries will contain a snapshot of the primaries data at any point in time. This is similar to RDBMS replications and even HBase’s own multi-datacenter replication, however in a single cluster.

- On the read side, the client can detect whether the read is coming from up-to-date data or is stale data. Also, the client can issue reads with different consistency requirements on a per-operation basis to ensure its own semantic guarantees.
- The client can still observe edits out-of-order, and can go back in time, if it observes reads from one secondary replica first, then another secondary replica. There is no stickiness to region replicas or a transaction-id based guarantee. If required, this can be implemented later though.



To better understand the TIMELINE semantics, lets look at above diagram. Lets say that there are two clients, and the first one writes x=1 at first, then x=2 and x=3 later. As above, all writes are handled by the primary region replica. The writes are saved in the write ahead log (WAL), and replicated to the other replicas asynchronously. In the above diagram, notice that replica_id=1 received 2 updates, and it's data shows that x=2, while the replica_id=2 only received a single update, and its data shows that x=1.

If client1 does reads with STRONG consistency, it will only talk with the replica_id=0, and thus guaranteed to observe the latest value for x=3. In case of a client issuing TIMELINE consistency, the RPC will go to all replicas (after primary timeout) and the result from the first response will be returned back. Thus the client can see either 1, 2 or 3 as the value of x. Let's say that the primary region has failed and log replication cannot continue for some time. If the client does multiple reads with TIMELINE consistency, she client can observe x=2 first, then x=1, and so on.

Tradeoffs

Having secondary regions hosted for read availability comes with some tradeoffs which should be carefully evaluated per use case. The main advantages of this design are

- High availability for read-only tables.
- High availability for stale reads
- Ability to do very low latency reads with very high percentile (99.9%+) latencies for stale reads

The downsides for this feature are

- Double / Triple memstore usage (depending on region replicaion count)
- Increased block cache usage
- Extra network traffic for log replication
- Extra backup RPCs for replicas

To serve the region data from multiple replicas, HBase opens the regions in secondary mode in the region servers. The regions opened in secondary mode will share the same data files with the primary region replica, however each secondary region replica will have its own memstore to keep the unflushed data (only primary region can do flushes). Also to serve reads from secondary regions, the blocks of data files may be also cached in the block caches for the secondary regions.

Configuration properties

To use high available reads, you should set the following properties in hbase-site.xml file.

Server side properties

```
  <property>
    <name>hbase.regionserver.storefile.refresh.period</name>
    <value>0</value>
    <description>
      The period (in milliseconds) for refreshing the store files for the
secondary regions. 0 means this feature is disabled. Secondary regions sees
new files (from flushes and compactions) from primary once the secondary
region refreshes the list of files in the region (there is no notification
```

```
mechanism). But too frequent refreshes might cause extra Namenode pressure. If
the files cannot be refreshed for longer than HFile TTL
(hbase.master.hfilecleaner.ttl) the requests are rejected. Configuring HFile
TTL to a larger value is also recommended with this setting.
    </description>
</property>

 <property>
    <name>hbase.master.loadbalancer.class</name>
 <value>org.apache.hadoop.hbase.master.balancer.StochasticLoadBalancer</value>
    <description>
      Only StochasticLoadBalancer is supported for using region replicas
    </description>
</property>
```

Client side properties

Ensure to set the following for all clients (and servers) that will use region replicas.

```
<property>
    <name>hbase.ipc.client.allowsInterrupt</name>
    <value>true</value>
    <description>
      Whether to enable interruption of RPC threads at the client side. This
is required for region replicas with fallback RPC’s to secondary regions.
    </description>
</property>
<property>
    <name>hbase.client.primaryCallTimeout.get</name>
    <value>10000</value>
    <description>
      The timeout (in microseconds), before secondary fallback RPC’s are
submitted for get requests with Consistency.TIMELINE to the secondary replicas
of the regions. Defaults to 10ms. Setting this lower will increase the number
of RPC’s, but will lower the p99 latencies.
    </description>
</property>
<property>
    <name>hbase.client.primaryCallTimeout.multiget</name>
    <value>10000</value>
    <description>
      The timeout (in microseconds), before secondary fallback RPC’s are
submitted for multi-get requests (HTable.get(List<Get>)) with
Consistency.TIMELINE to the secondary replicas of the regions. Defaults to
10ms. Setting this lower will increase the number of RPC’s, but will lower the
p99 latencies.
    </description>
```

```
</property>
```

Creating a table with region replication

Shell

```
create 't1', 'f1', {REGION_REPLICATION => 2}

describe 't1'
for i in 1..100
put 't1', "r#{i}", 'f1:c1', i
end
flush 't1'
```

Java

```
HTableDescriptor htd = new HTableDesctiptor(TableName.valueOf("test_table"));
htd.setRegionReplication(2);
...
admin.createTable(htd);
```

You can also use `setRegionReplication()` and alter table to increase, decrease the region replication for a table.

Region splits and merges

Region splits and merges are not compatible with regions with replicas yet. So you have to pre-split the table, and disable the region splits. Also you should not execute region merges on tables with region replicas. To disable region splits you can use `DisabledRegionSplitPolicy` as the split policy.

User Interface

In the masters user interface, the region replicas of a table are also shown together with the primary regions. You can notice that the replicas of a region will share the same start and end keys and the same region name prefix. The only difference would be the appended replica_id (which is encoded as hex), and the region encoded name will be different. You can also see the replica ids shown explicitly in the UI.

API and Usage

Shell

You can do reads in shell using a the Consistency.TIMELINE semantics as follows

```
hbase(main):001:0> get 't1','r6', {CONSISTENCY => "TIMELINE"}
```

You can simulate a region server pausing or becoming unavailable and do a read from the secondary replica:

```
$ kill -STOP <pid or primary region server>

hbase(main):001:0> get 't1','r6', {CONSISTENCY => "TIMELINE"}
```

Using scans is also similar

```
hbase> scan 't1', {CONSISTENCY => 'TIMELINE'}
```

Java

You can set set the consistency for Gets and Scans and do requests as follows.

```
Get get = new Get(row);
get.setConsistency(Consistency.TIMELINE);
…
Result result = table.get(get);
```

You can also pass multiple gets:

```
Get get1 = new Get(row);
get1.setConsistency(Consistency.TIMELINE);
…
ArrayList<Get> gets = new ArrayList<Get>();
gets.add(get1);
...
Result[] results = table.get(gets);
```

And Scans:

```
Scan scan = new Scan();
scan.setConsistency(Consistency.TIMELINE);
...
ResultScanner scanner = table.getScanner(scan);
```

You can inspect whether the results are coming from primary region or not by calling the Result.isStale() method:

```
Result result = table.get(get);
if (result.isStale()) {
  ...
}
```

Resources

1. More information about the design and implementation can be found at the jira issue: https://issues.apache.org/jira/browse/HBASE-10070

2. HBaseCon 2014 talk also contains some details and slides http://hbasecon.com/sessions/#session15