Replication In HBase

Yifan Gu
COSC-580

***Abstract*—The replication feature of Apache HBase provides a way to copy data between HBase deployments. It can serve as a disaster recovery solution and can contribute to provide higher availability at the HBase layer. The basic architecture pattern used for Apache HBase replication is master-push. One master cluster can replicate to any number of slave clusters, and each region server will participate to replicate their own stream of edits.**

Keywords—replication; consistency; HBase;

I. INTRODUCTION

Apache HBase is an open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable [1]. As Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS.

The replication feature of Apache HBase provides a way to copy data between HBase deployments. It can serve as a disaster recovery solution and can contribute to provide higher availability at the HBase layer. It can also serve more practically; for example, as a way to easily copy edits from a web-facing cluster to a "MapReduce" cluster which will process old and new data and ship back the results automatically.

The basic architecture pattern used for Apache HBase replication is master-push; it is much easier to keep track of what's currently being replicated since each region server has its own write-ahead-log (WAL or HLog), highly similar to other well-known solutions like MySQL master/slave replication where there's only one bin log to keep track of. One master cluster can replicate to any number of slave clusters, and each region server will participate to replicate their own stream of edits.

The replication is done asynchronously, meaning that the clusters can be geographically distant, the links between them can be offline for some time, and rows inserted on the master cluster won't be available at the same time on the slave clusters. Asynchronous replication implies eventual consistency.

The replication format used in this design is conceptually the same as MySQL's statement-based replication . Instead of SQL statements, whole WALEdits (consisting of multiple cell inserts coming from the clients' Put and Delete) are replicated in order to maintain atomicity.

The HLogs from each region server are the basis of HBase replication, and must be kept in HDFS as long as they are needed to replicate data to any slave cluster. Each RS reads from the oldest log it needs to replicate and keeps the current position inside ZooKeeper to simplify failure recovery. That position can be different for every slave cluster, same for the queue of HLogs to process.

The clusters participating in replication can be of asymmetric sizes and the master cluster will do its "best effort" to balance the stream of replication on the slave clusters by relying on randomization.

II. HBASE

Apache HBase is an open-source, distributed, versioned, column-oriented store modeled. It provides Bigtable-like capabilities on top of Hadoop and HDFS [2].

A. Write Path

Apache HBase is the Hadoop database, and is based on the Hadoop Distributed File System (HDFS). HBase makes it possible to randomly access and update data stored in HDFS. The write path begins at a client, moves to a region server, and ends when data eventually is written to an HBase data file called an Hfile (Figure 1).

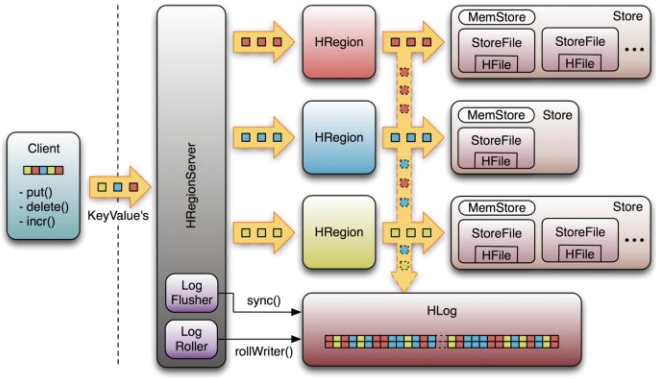

Figure 1: Hbase Write Path

HBase data is organized similarly to a sorted map, with the sorted key space partitioned into different shards or regions. An HBase client updates a table by invoking put or delete commands. When a client requests a change, that request is routed to a region server right away by default. However, programmatically, a client can cache the changes in the client side, and flush these changes to region servers in a batch, by turning the autoflush off..

Since the row key is sorted, it is easy to determine which region server manages which key. A change request is for a specific row. Each row key belongs to a specific region which is served by a region server. So based on the put or delete's key, an HBase client can locate a proper region server. At first, it locates the address of the region server hosting the -ROOT-

region from the ZooKeeper quorum. From the root region server, the client finds out the location of the region server hosting the -META- region. From the meta region server, then we finally locate the actual region server which serves the requested region. This is a three-step process, so the region location is cached to avoid this expensive series of operations. If the cached location is invalid, it's time to re-locate the region and update the cache.

After the request is received by the right region server, the change cannot be written to a HFile immediately because the data in a HFile must be sorted by the row key. This allows searching for random rows efficiently when reading the data. Data cannot be randomly inserted into the HFile. Instead, the change must be written to a new file. If each update were written to a file, many small files would be created. Such a solution would not be scalable nor efficient to merge or read at a later time. Therefore, changes are not immediately written to a new HFile.

Instead, each change is stored in a place in memory called the memstore, which cheaply and efficiently supports random writes. Data in the memstore is sorted in the same manner as data in a HFile. When the memstore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. Completing one large write task is efficient and takes advantage to HDFS' strengths [3].

Although writing data to the memstore is efficient, it also introduces an element of risk: Information stored in memstore is stored in volatile memory, so if the system fails, all memstore information is lost. To help mitigate this risk, HBase saves updates in a write-ahead-log (WAL) before writing the information to memstore. In this way, if a region server fails, information that was stored in that server's memstore can be recovered from its WAL.

The data in a WAL file is organized differently from HFile. WAL files contain a list of edits, with one edit representing a single put or delete. The edit includes information about the change and the region to which the change applies. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk. Because WAL files are ordered chronologically, there is never a need to write to a random place within the file [4].

As WALs grow, they are eventually closed and a new, active WAL file is created to accept additional edits. This is called "rolling" the WAL file. Once a WAL file is rolled, no additional changes are made to the old file.

B. *Replication Overview*

The underlying principle of HBase replication is to replay all the transactions from the master to the slave. This is done by replaying the WALEdits in the WALs from the master cluster. These WALEdits are sent to the slave cluster region servers, after filtering (whether a specific edit is scoped for replication or not) and shipping in a customized batch size (default is 64MB). In case the WAL Reader reaches the end of the current WAL, it will ship whatever WALEdits have been read till then (Figure 2). Since this is an asynchronous mode of replication, the slave cluster may lag behind from the master in a write heavy application by the order of minutes.

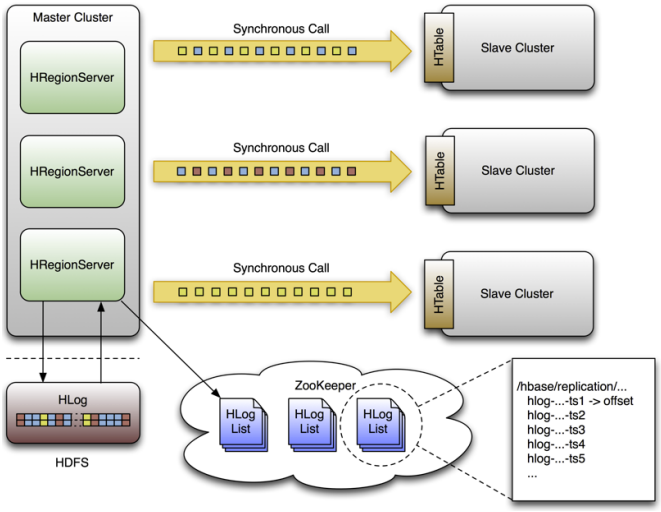

Figure 2: Replication Overview

C. *ClusterID*

Every HBase cluster has a clusterID, a UUID type auto generated by HBase. It is kept in underlying filesystem (usually HDFS) so that it doesn't change between restarts. This is stored inside the /hbase/hbaseid znode. This id is used to acheive master-master/acyclic replication. A WAL contains entries for a number of regions which are hosted on the regionserver. The replication code reads all the keyvalues and filters out the keyvalues which are scoped for replication. It does this by looking at the column family attribute of the keyvalue, and matching it to the WALEdit's column family map data structure. In the case that a specific keyvalue is scoped for replication, it edits the clusterId parameter of the keyvalue to the HBase cluster Id.

III. ZOOKEEPER

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming [5].

The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available (Figure 3).

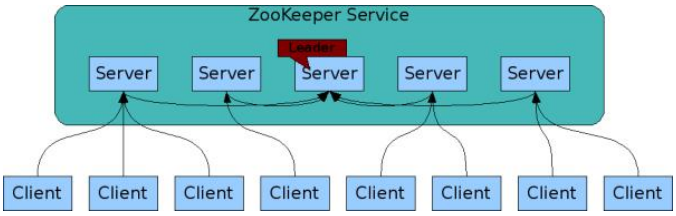

Figure 3: ZooKeeper Service

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server [6].

Zookeeper plays a key role in HBase Replication, where it manages and coordinates almost all the major replication activity, such as registering a slave cluster, enqueuing new WALs, handling region server failover, etc.

A. *Hierarchical Namespace*

The name space provided by ZooKeeper is much like that of a standard file system. A name is a sequence of path elements separated by a slash (/). Every node in ZooKeeper's name space is identified by a path (Figure 4).

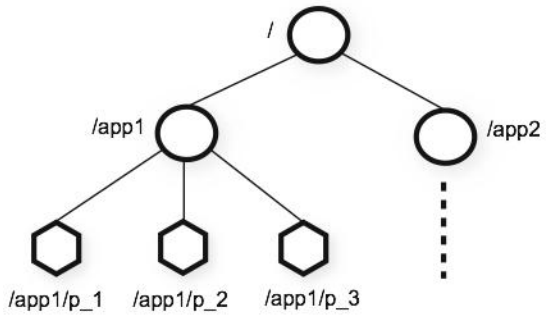

Figure 4: ZooKeeper Hierarchical Namespace

B. *Replication Zookeeper State*

HBase replication maintains all of its state in Zookeeper. By default, this state is contained in the base znode:

/hbase/replication

There are three major child znodes in the base replication znode:

- **State znode:** /hbase/replication/state

  The state znode indicates whether or not replication is enabled on the cluster corresponding to this zookeeper quorum.

  /hbase/replication/state [VALUE: true]

- **Peers znode:** /hbase/replication/peers

  The peers znode contains a list of all peer replication clusters and the current replication state of those clusters. Each of these peer znodes has a child znode that indicates whether or not replication is enabled on that peer cluster.

  /hbase/replication/peers

  /1/peer-state [Value: ENABLED]

  /2/peer-state [Value: DISABLED]

- **RS znode:** /hbase/replication/rs

  The rs znode contains a list of all outstanding HLog files in the cluster that need to be replicated. The list is divided into a set of queues organized by region server and the peer cluster the region server is shipping the HLogs to. The rs znode has one child znode for each region server in the cluster. The child znode name is simply the regionserver name (a concatenation of the region server's hostname, client port and start code). These region servers could either be dead or alive.

  /hbase/replication/rs

  /hostname.example.org,6020,1234

  /hostname2.example.org,6020,2856

  Within each region server znode, the region server maintains a set of HLog replication queues. Each region server has one queue for every peer cluster it replicates to. These queues are represented by child znodes named using the cluster id of the peer cluster they represent.

  Each queue has one child znode for every HLog that still needs to be replicated. The value of these HLog child znodes is the latest position that has been replicated. This position is updated every time a HLog entry is replicated.

  /hbase/replication/rs

  /hostname.example.org,6020,1234

  /1

  23522342.23422 [VALUE: 254]

  12340993.22342 [VALUE: 0]

IV. REPLICATION

The replication feature of Apache HBase provides a way to copy data between HBase deployments. It can serve as a disaster recovery solution and can contribute to provide higher availability at the HBase layer.

A. *Replication Modes*

There are three modes for setting up HBase Replication:

- **Master-Slave:**

  In this mode, the replication is done in a single direction, i.e., transactions from one cluster are pushed to other cluster.

- **Master-Master:**

  In this mode, replication is sent across in both the directions, for different or same tables, i.e., both the clusters are acting both as master and slave. In the case that they are replicating the same table, it may lead to a never ending loop, but this is avoided by setting the cluster ID of a Mutation (Put/Delete) to the cluster ID of the originating cluster.

- **Cyclic:**

  In this mode, there are more than two HBase clusters that are taking part in replication setup, and one can have various possible combinations of master-slave and

master-master set up between any two clusters. The above two covers those cases well; one corner situation is when we have set up a cycle.

B. Replication Source

The Replication Source is a java Thread object in the region server process and is responsible for replicating WAL entries to a specific slave cluster. It has a priority queue which holds the log files that are to be replicated. As soon as a log is processed, it is removed from the queue. The priority queue uses a comparator that compares the log files based on their creation timestamp. Therefore, logs are processed in the same order as their creation time (older logs are processed first). If there is only one log file in the priority queue, it will not be deleted as it represents the current WAL.

C. Edits Operation

By default, a source will try to read from a log file and ship log entries as fast as possible to a sink. This is first limited by the filtering of log entries; only Key Values that are scoped GLOBAL and that don't belong to catalog tables will be retained. A second limit is imposed on the total size of the list of edits to replicate per slave, which by default is 64MB. This means that a master cluster's region server with 3 slaves will use at most 192MB to store data to replicate [7].

Once the maximum size of edits was buffered or the reader hits the end of the log file, the source thread will stop reading and will choose at random a sink to replicate to. It will directly issue a RPC to the chosen machine and will wait for the method to return. If it's successful, the source will determine if the current file is emptied or if it should continue to read from it. If the former, it will delete the znode in the queue. If the latter, it will register the new offset in the log's znode. If the RPC threw an exception, the source will retry 10 times until trying to find a different sink [8].

D. Logs Cleaning

If replication isn't enabled, the master's logs cleaning thread will delete old logs using a configured TTL. This doesn't work well with replication since archived logs passed their TTL may still be in a queue. Thus, the default behavior is augmented so that if a log is passed its TTL, the cleaning threads will look up every queue until it finds the log (while caching the ones it finds). If it's not found, the log will be deleted. The next time it has to look for a log, it will first use its cache.

E. Normal Processing

The client uses an API that sends a Put, Delete or ICV to a region server. The key values are transformed into a WALEdit by the region server and is inspected by the replication code that, for each family that is scoped for replication, adds the scope to the edit. The edit is appended to the current WAL and is then applied to its MemStore.

In a separate thread, the edit is read from the log and only the KVs that are replicable are kept.

The edit is then tagged with the master's cluster UUID. When the buffer is filled, or the reader hits the end of the file, the buffer is sent to a random region server on the slave cluster.

Synchronously, the region server that receives the edits reads them sequentially and separates each of them into buffers, one per table. Once all edits are read, each buffer is flushed using the normal HBase client. This is done in order to leverage parallel insertion. The master's cluster UUID is retained in the edits applied at the slave cluster in order to allow cyclic replication.

Back in the master cluster's region server, the offset for the current WAL that's being replicated is registered in ZooKeeper.

F. Non-responding slave clusters

In the separate thread, the region server reads, filters and buffers the log edits the same way as during normal processing. The slave region server that's contacted doesn't answer to the RPC, so the master region server will sleep and retry up to a configured number of times. If the slave RS still isn't available, the master cluster RS will select a new subset of RS to replicate to and will retry sending the buffer of edits.

In the meantime, the WALs will be rolled and stored in a queue in ZooKeeper. Logs that are archived by their region server will update their paths in the in-memory queue of the replicating thread.

When the slave cluster is finally available, the buffer will be applied the same way as during normal processing. The master cluster RS will then replicate the backlog of logs.

G. Region server failover

As long as region servers don't fail, keeping track of the logs in ZK doesn't add any value. Unfortunately, they do fail, so since ZooKeeper is highly available we can count on it and its semantics to help us managing the transfer of the queues.

All the master cluster region servers keep a watcher on every other one of them to be notified when one dies (just like the master does). When it happens, they all race to create a znode called "lock" inside the dead RS' znode that contains its queues. The one that creates it successfully will proceed by transferring all the queues to its own znode and will delete all the old ones when it's done. The recovered queues' znodes will be named with the id of the slave cluster appended with the name of the dead server.

Once that is done, the master cluster RS will create one new source thread per copied queue, and each of them will follow the read/filter/ship pattern. The main difference is that those queues will never have new data since they don't belong to their new region server, which means that when the reader hits the end of the last log, the queue's znode will be deleted and the master cluster RS will close that replication source.

V. IMPLEMENTATION AND EXPERIMENTS

HBase replication provides a means of copying the data from one HBase cluster to another HBase cluster. The cluster receiving the data from user applications is called the master

cluster, and the cluster receiving the replicated data from the master is called the slave cluster.

OS	Windows 7
Virtual Machine	VMware workstation 8.0.4
Number of Servers	3 (Bt-1, Bt-2, Bt-3)
Server OS	BackTrack 5 RC3
Hadoop	1.0.4
ZooKeeper	3.4.5
Hbase Foundation	0.94.6

Table 1: Software Deployment Information

The experiment uses VMware to emulate the two clusters (Table 1): cluster-1 (server bt-1) and cluster-2 (server bt-2 and server bt-3). Servers among clusters can communicate with each other via SSH (Figure 5).

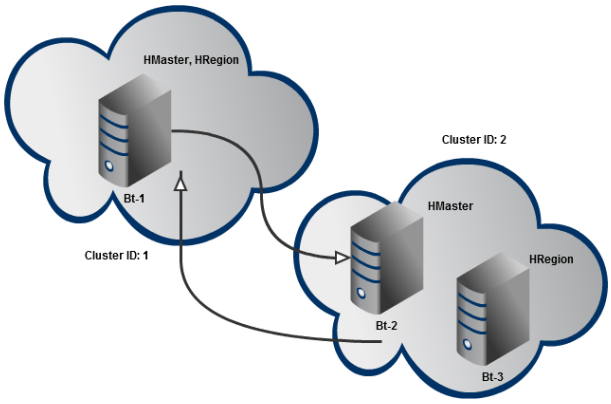

Figure 5: Servers Layout

Cluster 1 only contains one server Bt-1, which serves simultaneously as Hbase Master and Hbase Region Server(Table 2).

Server	HBase	Cluster ID	Znode.parent
Bt-1	HMaster, HRegion	1	/Hbase
Bt-2	HMaster	2	/Hbase-2
Bt-3	HRegion	2	/Hbase-2

Table 2: Servers Layout Information

Three servers live in one ZooKeeper cluster, where cluster-1's parent znode is Hbase and cluster-2's parent znode is Hbase-2. Every table that contains families that are scoped for replication must exist on each cluster and have exactly the same name.

Row	State (SCOPE = GLOBAL)	City(SCOPE = LOCAL)
No1	VA	Arlington
No2	NY	NYC
No3		Fairfax
No4	MA	

Table 3: Content of Table 'LOCATION' in Cluster 2

In this experiment, cluster-1 and cluster-2 both firstly create a table "LOCATION" , column "State" which is scoped for 'GLOBAL', and column "City" which is scoped for 'LOCAL'. Then, Bt-2 in Cluster-2 puts a series of data into Hbase. Because Bt-3 lives in the same cluster with Bt-2, they see the same sequence and content of stored data (Table 3).

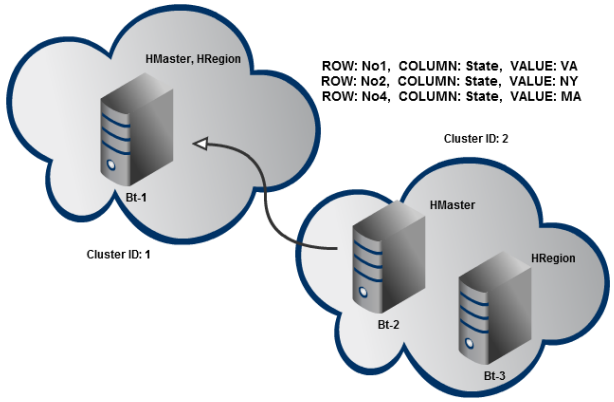

Figure 6: Edits Shipping in Servers Layout

However, only Key Values that are scoped GLOBAL and that don't belong to catalog tables will be retained (Figure 6). Therefore, edits shipping to cluster-1 only include Row Key "No1", "No2", and "No4" (Table 4).

Row	State (SCOPE = GLOBAL)	City(SCOPE = LOCAL)
No1	VA(REPLICATED)	
No2	NY(REPLICATED)	
No4	MA(REPLICATED)	

Table 4: Content of Table 'LOCATION' in Cluster 1

Additionally, because all edits are time stamped and the storage system supports storage and retrieval of multiple versions of a cell, applications can query for the latest version according to timestamp and Cluster-1 sees the same sequence and content with Cluster-2. Moreover, although replication process can get the same results and run successfully when any region server fails and restarts, the process cannot continue when HMaster has failure.

CONCLUSION

The replication feature of Apache HBase provides a way to copy data between HBase deployments. It can serve as a disaster recovery solution and can contribute to provide higher availability at the HBase layer. It can be used for a variety of purposes beyond the obvious disaster recovery. While there may have some pitfalls with the current implementation, they are minor when compared to the gains.

ACKNOWLEDGMENT

The author would like to thank Prof. Wenchao Zhou for his instructions and comments and thank Apache Software Foundation and Hbase collaborators for providing the fabulous open source software platform and resources.

REFERENCES

[1] Reza Moraveji , Javid Taheri , Mohammad Reza , Nikzad Babaii Rizvandi , Albert Y. Zomaya, "Data-Intensive Workload Consolidation for the Hadoop Distributed File System", Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing, p.95-103, September 20-23, 2012.

[2] L. George, Hbase: The Definitive Guide, O'Reilly Media, Inc, 2011

[3] Apache Hadoop. http://hadoop.apache.org/

[4] Apache Hbase. http://hbase.apache.org/

[5] Apache ZooKeeper. http://zookeeper.apache.org/

[6] P. Hunt, M. Konar, F. P. Junqueira and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems", Proceedings of the 2010 USENIX conference on USENIX annual technical conference, p 11-11, 2010

[7] M. N. Vora, "Hadoop-HBase for large-scale data," Computer Science and Network Technology (ICCSNT), 2011 International Conference on , vol.1, no., pp.601,605, 24-26 Dec. 2011

[8] D. Carstoiu, A. Cernian, A. Olteanu., "Hadoop Hbase-0.20.2 performance evaluation," New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on , vol., no., pp.84,87, 11-13 May 2010