User-based Load Balancer in HBase

Ahmad Ghandour[1], Mariam Moukalled[1], Mohamad Jaber[1] and Yliès Falcone[2]
[1]*American University of Beirut, Computer Science Department, Beirut, Lebanon*
[2]*Univ. Grenoble Alpes, Inria, Laboratoire d'Informatique de Grenoble, F-38000 Grenoble, France*

Keywords: Big Data, NoSQL, HBase, Load Balancer.

Abstract: Latency of read and write operations is an important measure in relational and non-relational databases. Load balancing is one of the features that manages the distribution of the load between nodes in case of distributed servers to improve the overall performance. In this paper, we introduce a new load balancer to HBase (a non-relation database), which monitors the most requested keys and dynamically acts to redistribute the regions by splitting and moving them. Our load balancer takes into account the average response time of clients' requests and the most requested keys. Our method is fully implemented and can be integrated in HBase distribution. Experimental results show that we get on average an improvement of latency of 15%, and up to 35% in some scenarios.

1 INTRODUCTION

We live in the big data era. A tremendous amount of data, petabytes in size, are accessed by users. In standard relational database management systems, handling a huge amount of data in a reasonable amount of time became a bottleneck. Standard relational database management systems suffer from handling huge volume of data in reasonable amount time. This is mainly due to its relational structure of data. To this end, non-relational databases (NoSQL) were proposed to provide a mechanism to efficiently store and retrieve big data, which is no longer modeled with the tabular relations used in relational databases.

HBase is a NoSQL database. It is an open source implementation of Big Table (Chang et al., 2008). Big Table was proposed by Google to handle big data using distributed file system storage, Google File System (Ghemawat et al., 2003), in well-formed structure. The main goals of NoSQL databases are wide applicability, scalability, high performance and high availability. Contrarily to relational databases, NoSQL uses a simple data model which allows for dynamic control over data layout and format.

NoSQL databases are built on top of a distributed file system storage. For instance, HBase stores its data in Hadoop Distributed File System (HDFS) (Hadoop, 2016). HDFS consists of distributed nodes where data is stored. However, HDFS only allows for sequential access to the files which requires reading block of data to access a single row, while HBase is built on top of it to allow random file access, and hence efficient handling of read/write operations.

In HBase a table is a collection of rows, where the rows are sorted according to keys. The table schema defines only column families, which are the key-value pairs. A table may have multiple column families and each column family can have any number of columns. Additionally, each cell value of the table has a timestamp.

Hbase system is composed of several components that formulate a hierarchical structure: (1) on the top level there are Zookeeper and HMaster; (2) in the middle level there are region servers and regions; and (3) finally there are the store, store file, HFile and MemStore. Zookeeper provides coordination services for distributed applications such as naming, configuration management, synchronization, and group services. It consists of several servers. One of the server acts as the master and the others as replicas. Zookeeper stores all the META data, which is composed of all region servers and their data start and end keys. It also coordinates with the HMaster to update its META data accordingly. Moreover, it handles all the clients' requests to locate the region servers given the requested key. HMaster is a centralized node that handles the management of its region servers (e.g., crash of servers, new servers). It also communicates with Zookeeper to update META data information, and it is responsible for the assignment of regions

Ghandour, A., Moukalled, M., Jaber, M. and Falcone, Y.
User-based Load Balancer in HBase.
DOI: 10.5220/0006290103920396
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 364-368
ISBN: 978-989-758-243-1

among region servers and re-assignment of regions due to failure or load balancing. Region servers store regions. Each region server has a start and an end key except for the first region has empty start key. Region server handles all read/write requests for all its regions. Regions act as the container for one or more store. Each region has a maximum size of 10GB and it can be manually configured with smaller sizes. A store in the region has a minimum size of 64MB and it can be configured with larger sizes. Store contains *MemStore* and *HFiles*. MemStore acts as a cache memory, which initially stores any data written. When the MemStore is full, it is flushed and the data is transferred to HFiles.

HBase supports several load balancing techniques to equally split the work amongst the region servers. Nonetheless, all the load balancers implemented in HBase do not take into account the response time of the users' requests. Additionally, the existing load balancers split the regions in the middle and do not take into account the hot keys (i.e., keys requested so frequently) by the clients.

In this paper, we introduce a new balancer of HBase region servers, which takes into account the response time of the users' requests and automatically detects regions causing delay in response times. Then, it splits those regions in order to dispatch the users' requests into the new created regions. Moreover, if some region servers have too many (with respect to some threshold) regions, it moves regions from high-loaded region servers to other less-loaded region servers.

The remainder of the paper is structured as follows. In section 2, we present our main method to load balance the distribution of the regions. Section 3 shows experimental results. In Section 4, we discuss related work. Finally, Section 5 draws some conclusions and perspectives.

2 ADAPTIVE HBase

In this section, we introduce an efficient user-based balancing method, which automatically splits and moves HBase regions to improve response time of clients' requests. The method mainly consists of two phases. First, we integrate a monitoring feature to the APIs provided by HBase that allow clients to access data. Second, we implement an algorithm to automatically detect regions causing overhead, and accordingly regions are split or moved to improve the performance of the clients' request.

2.1 Runtime Monitoring and Profiling

The first step is to detect regions causing overhead. For this, we intercept the HBase API to monitor the load of the region servers. The interception allows to keep track of the keys accessed per regions, the total number of accesses per key and the average response time with respect to each region. The profiling data is stored in the META table. Consequently, whenever a client requests to get any key using shell-based or API, the new integrated code allows to monitor and store the required information to the META table. Note that the META table remains on one of the region servers and it does not split regardless of its size. Moreover, the new column family containing the logs is cleared each time we execute the balancer, which is defined in the following.

2.2 Runtime User-based Load Balancing

We implement a user-based load balancer, which dynamically distributes the load among the servers according to client requests. We define a *CronJob*, a time-based scheduler, to periodically run the user-based balancer at specific time instants (to be specified by the HBase administrator). The user-based balancer is split into three phases.

2.2.1 Select Victim Regions

The first phase consists in selecting victim regions according to the following:

1. Compute, from the profiled information, the average response time per region. Let $ART = \{(R_1, art_1), \ldots, (R_n, art_n)\}$, where art_i is the average respond time of region R_i.
2. Select the top k victim regions VRs with average response time greater than some threshold, $ART^{\texttt{threshold}}$. $ART^{\texttt{threshold}}$ is an input parameter of the algorithm to be specified by the HBase administrator depending on the clients' need and the specification of the cluster. Formally, $VRs = \{R_i \mid (R_i, art_i) \in ART \wedge art_i \geq ART^{\texttt{threshold}}\}$.

2.2.2 Split Phase

The second phase consists in splitting victim regions with respect to some keys. A region has to be split so that the requests are dispatched over the two new regions. That is, consider the case where there is a massive number of requests on two keys k_1 and k_2 in some region, where the two keys belong to the first part of the region. Then, it is not desirable to split the region

in the middle, since it will remain congested. For this, we split the region with respect to a *balance key* so that after the split, the future requests would be eventually dispatched between the two new regions. Note that small regions should not be split to avoid explosion of regions, but they should be moved to another less loaded region server, which will be discussed in the next phase. For this, we only split the region if its size is greater than $RS^{\texttt{threshold}}$. For every victim region $vr \in VRs$ of size is greater than $RS^{\texttt{threshold}}$, the split phase is defined as follows:

1. Let $(k_1, r_1), \ldots, (k_\alpha, r_\alpha)$ be the number of requests, sorted with respect to the keys in the victim region vr.
2. We define the balance key to be equal to one if α is equal to one (i.e., only one key is requested by this region). Otherwise, we define the balance key, bk, so that after the split the requests would be split into the two new regions. For this, we first select the first key, k_β, from the requested keys satisfying the following property. The number of requests to the keys less than or equal to k_β would be greater than the number of requests to the keys greater than k_β. Then, we set the balance key to be between k_β and $k_{\beta+1}$. Formally, $bk = \frac{k_\beta + k_{\beta+1}}{2}$, where β is the index of the key such that $\sum_{i=1}^{i=\beta} r_i \leq r^{av}$ and $\sum_{i=1}^{i=\beta+1} r_i > r^{av}$ and $r^{av} = \frac{\sum_{i=1}^{i=\alpha} r_i}{2}$.

2.2.3 Move Phase

After the split phase some region servers may have many regions. For this, for every region server that corresponds to some victim regions, we count its number of regions. If the number of regions is greater than some threshold, $NR^{\texttt{threshold}}$, we move the extra regions to other region servers with minimum number of regions and less than $NR^{\texttt{threshold}}$. $NR^{\texttt{threshold}}$ is an input parameter of our algorithm, which depends on the capacity of region servers.

2.3 General Algorithm

The general structure of the algorithm of the user-balancer is depicted in Listing 1. It mainly consists of three phases:

1. Select victim regions.
2. Splitting of victim regions.
3. Moving of regions from highly-loaded region servers to less-loaded region servers.

Table 1: Execution Times (Seconds) - 1 client - 2 keys.

Requests	RT Before	RT After	Improvement
100,000	26.61	19.6	26.34%
250,000	52.39	48.85	6.76%
500,000	101.97	93.61	8.20%
750,000	153.64	140.10	8.81%
1,000,000	203.43	183.80	9.65%

Table 2: Execution times (seconds) - 1 client - 4 keys.

Requests	RT Before	RT After	Improvement
100,000	40.71	36.60	10.10%
250,000	97.70	88.86	9.05%
500,000	196.82	162.69	17.34%
750,000	300.52	245.01	18.47%
1,000,000	410.46	326.61	20.43%

Discussion. In case where the average response time is very high (i.e., greater than $ART^{\texttt{threshold}}$), the size of the region is smaller than $RS^{\texttt{threshold}}$, and the other region servers contain more regions than $NR^{\texttt{threshold}}$, the performance of the HBase cluster is not compatible with respect to user needs, and hence region servers have to be replaced with more powerful machines to satisfy the users' needs.

3 EXPERIMENTAL RESULTS

We evaluate our algorithm on a Hadoop cluster consisting of 8 nodes with 8 cores each. We populate the HBase with a table consisting of several regions, each of size 4GB.

We implement several scenarios: (1) one client requesting two keys; (2) one client requesting four keys; (3) four clients requesting two keys; and (4) four clients requesting four keys. For each scenario, we vary the number of requests from $100,000$ to $1,000,000$ and we measure the execution times to handle them.

The threshold parameters were defined according to the cluster configuration: (1) $AR^{\texttt{threshold}}$, i.e., average response time per request, is set to be 0.2ms; and (2) $RS^{\texttt{threshold}}$ to be 1GB. Tables 1, 2, 3 and 4, show the execution times before and after running our user-based load balancer. Note that, the execution times of monitoring are included when running our balancer. In case of requesting two keys by one clients we get an average improvement of 11.95%. In case of requesting four different keys we get an average improvement of 15.08%. In case of four clients, we get an improvement of 37.59% in case of requesting two keys each with $250,000$ hits. Moreover, as average we get an improvement 15.16%. In case of four clients requesting four different keys, we get an

Listing 1: User-Based Balancer Algorithm.

```
/*** Select Victim Regions ***/
ART = averageRTPerRegion();
VRs = filterVitimRegions(ART);

/*** Split Phase ***/
for(vr: VRs) {
   if( size(vr) < RS_THRESHOLD ) continue;
   TopKKeys = getTopKKeys(vr);
   if( size(TokKKeys) == 1 ) {
      balanceKey = middleKey(vr);
   } else {
      requestAverage = sumRequests(TokKKeys);
      betaIndex = computeBetaIndex(TopKKeys);
      balanceKey = (TopKKeys[betaIndex] + TopKKeys[betaIndex + 1]) / 2;
   }
   split(vr, balanceKey);
}

/*** Move Phase ***/
for(vr: VRs) {
   RS = regionServer(vr);
   if( numberRegions(RS) <= NR_THRESHOLD ) continue;
   count = numberRegions(RS) - NR_THRESHOLD;
   RM = selectRegionsToMove();
   move(count, vr, RM);
}
```

average improvement of 9.99%.

4 RELATED WORK

4.1 Native HBase Load Balancing

HBase supports several types of load balancing. By default, it runs the stochastic load balancer periodically every 5 minutes. Below is the description of different balancers supported by HBase:

- Simple load balancer (Simple Load Balancer, 2016) computes the average number of regions and iterates through the most loaded servers to redistribute the regions on the less loaded servers (i.e., less number of regions).
- Favored node load balancer (Favored Node Load Balancer, 2016) takes into consideration server failure, so that when a server fails the regions will allocate to the less favorable server, which is defined in HDFS.
- Stochastic Load Balancer (Stochastic Load Balancer, 2016) is based on the cost of region load, table load, data locality, MemStore sizes, store file sizes and change the state of the cluster.
- Capacity-aware load balancer unlike other balancers takes into consideration the capacity and the characteristics of the nodes and distribute the load according to the capacities of the machines.
- Table region balancer is based on distributing table equally on all region servers. This balancer results in good performance when clients requests the data from same table.

To the best of our knowledge none of the above balancers consider: (1) the average response time of clients' requests and; (2) the number of requests per keys, where we can perform an efficient split based on balanced keys as presented in our algorithm.

Table 3: Execution times (seconds) - 4 clients - 2 keys.

Requests	RT Before	RT After	Improvement
100,000	40.45	35.48	12.29%
250,000	133.23	83.15	37.59%
500,000	178.17	164.58	7.63%
750,000	263.9	244.48	7.36%
1,000,000	365.54	325.56	10.94%

Table 4: Execution times (seconds) - 4 clients - 4 keys.

Requests	RT Before	RT After	Improvement
100,000	67.07	61.13	8.86%
250,000	167.96	151.37	9.88%
500,000	329.74	298.64	9.43%
750,000	511.57	448.59	12.31%
1,000,000	658.01	595.65	9.48%

4.2 Mixed Load Balancer

In Locality-Aware Load Balancer for HBase (Kewal Panchputre and Garg, 2016), they implemented an algorithm to combine different types of load balancers supported by HBase. The locality load balancer takes into consideration server balance, table balance and locality. The algorithm was tested on basics read write operations where it shows a better performance than simple load balancer. The proposed solution does not take into consideration neither the balance key to do optimized splits nor the average response time of clients' requests.

4.3 HBase Monitoring

Hannibal (Hannibal, 2016) introduced a monitoring tool to help HBase administrators to monitor and maintain HBase clusters that are configured for manual splitting. The proposed tool introduces monitoring technique for the region distribution on the cluster and region splits per table. Moreover, it displays the regions of a table ordered by size to help administrators making decision of splitting. Nonetheless, it does not provide an automatic split or move of the regions and the administrator has to manually distribute the regions based on the reported results.

4.4 Latency-based Optimization

The basic motivation of our algorithm was the solution proposed in (Sharov et al., 2015) to dynamically handle leader selection in distributed systems by monitoring client workload for previous time frames. Then, based on certain threshold and the location of the servers, the leader will be selected. The considered workload is the global load in the cluster and not the local one. Moreover, the clients are dynamically grouped according to their location and requests (read/write). The leader election algorithm is partitioned in two phases:

- Leader placement based on averaging latency of server operations. The latency is calculated based on the last time interval (1 day) and introducing a weight decay parameter to compute latency based on past intervals.
- Leader and Replica roles: in this tier the algorithm optimizes the voters selection in quorum based on replica location the selected leader.

5 CONCLUSION AND FUTURE WORK

We propose an algorithm that enhances client operation latency by monitoring and dynamically balancing (by splitting or moving regions) the region servers of HBase system based on the most requested keys and the average response time of clients' requests. The evaluation shows that our algorithm reduces the latency of client requests in case where some keys are highly requested.

For future work, we consider several directions. First, we want to extend our algorithm to take into account write operations. Second, we consider testing our algorithm on larger clusters and tables. Third, we want to introduce an automatic compaction of regions when the number of regions becomes too small.

REFERENCES

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2).

Favored Node Load Balancer (2016). Favored Node Load Balancer. `https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/master/balancer/FavoredNodeLoadBalancer.html`.

Ghemawat, S., Gobioff, H., and Leung, S. (2003). The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43.

Hadoop (2016). HDFS. `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

Hannibal (2016). Hannibal HBase. `https://github.com/sentric/hannibal`.

Kewal Panchputre, P. C. and Garg, R. (2016). Locality-aware load balancer for hbase. Technical report, University of Minnesota, Twin Cities.

Sharov, A., Shraer, A., Merchant, A., and Stokely, M. (2015). Take me to your leader! online optimization of distributed storage configurations. *PVLDB*, 8(12):1490–1501.

Simple Load Balancer (2016). Simple Load Balancer. `https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/master/balancer/SimpleLoadBalancer.html`.

Stochastic Load Balancer (2016). Stochastic Load Balancer. `https://hbase.apache.org/devapidocs/org/apache/hadoop/hbase/master/balancer/StochasticLoadBalancer.html`.