Load balancing for Key Value Data Stores

Ainhoa Azqueta-Alzúaz
Marta Patiño-Martínez
Universidad Politécnica de Madrid
Madrid, Spain
{aazqueta,mpatino}@fi.upm.es

Ivan Brondino
Ricardo Jimenez-Peris
LeanXcale
Madrid, Spain
{ivan.brondino,rjimenez}@leanxcale.com

ABSTRACT

In the last decade new scalable data stores have emerged in order to process and store the increasing amount of data that is produced every day. These data stores are inherently distributed to adapt to the increasing load and generated data. HBase is one of such data stores built after Google BigTable that stores large tables (hundreds of millions of rows) where data is stored sorted by key. A region is the unit of distribution in HBase and is a continuous range of keys in the key space. HBase lacks a mechanism to distribute the load across region servers in an automated manner. In this paper, we present a load balancer that is able to split tables into an appropriate number of regions of appropriate sizes and distribute them across servers in order to attain a balanced load across all servers. The experimental evaluation shows that the performance is improved with the proposed load balancer.

Keywords

Big Data, Key Value, HBase, Load Balancing, Performance

1. INTRODUCTION

During the last years new scalable data stores have emerged in order to process and store the increasing amount of data that is produced every day. These data stores also known as NoSQL data stores remove most of the relational databases properties in order to achieve high scalability. These data stores are inherently distributed to adapt to the increasing load and generated data. HBase [2] is one of such data stores built after Google BigTable that stores large tables (hundreds of millions of rows) where data is stored sorted by key. Each table defines a set of column families and a column can be defined at any time (two rows may have different columns). Data in HBase is organized into regions, which are the unit of distribution. A region is a continuous range of keys in the key space. HBase provides mechanisms for load balancing across servers by moving regions among servers. However, regions remain unchanged. That is, if a region becomes a hotspot, the HBase load balancing mechanisms will not distribute the region among the servers. That region will be managed by a single server. Another example where this may happen is the case when a region hosts much more keys than other regions (it manages more keys) and most of the load targets that region. Paper [4] proposes a load balancing algorithm for HBase. The algorithm moves regions across the servers to balance the load however, if a region becomes a hot spot and most of the load (for instance, 90% of the load) targets that region, moving the region would not balance the load among the servers. The only way to balance the load is to partition that region into smaller regions which then, will be moved to consecutive servers.

In this paper we target the partition of regions into regions hosting a similar number of rows. This policy is effective for those situations where a region becomes overloaded and by applying the predefined load balancing mechanisms the situation cannot improve. Our preliminary results loading the database defined by TPC-C benchmark for 3000 warehouses in a cluster of ten servers for storing data show that the throughput is increased one order of magnitude and the latency decreases two orders of magnitude.

The rest of the paper is organized as follows. Section 2 presents an introduction to HBase. Section 3 describes the proposed data partitioning algorithm. Section 4 describes the performance evaluation and the cost of the proposed approach. Finally, conclusions are presented in Section 5.

2. HBASE

HBase [2] is a sparse distributed scalable key-value data store modelled after Google BigTable [1].

HBase organizes data in very large tables with billions of rows and millions columns. Rows are uniquely identified by a key. Columns are organized into column families, which are defined at the time a table is created. Columns can be defined at any time and can vary across rows. A cell is a combination of {*key, column family, column*} and contains a *value* and a *timestamp* which represents the value version. Timestamps are automatically defined or can be user defined. For instance, the cell {customerid, address:home} references the last provided home address of the *customerid*, which is stored in the column family *address* and column *home*. Keys are bytes and rows are sorted alphabetically based on their key.

Tables are distributed in a cluster through regions. *Re-*

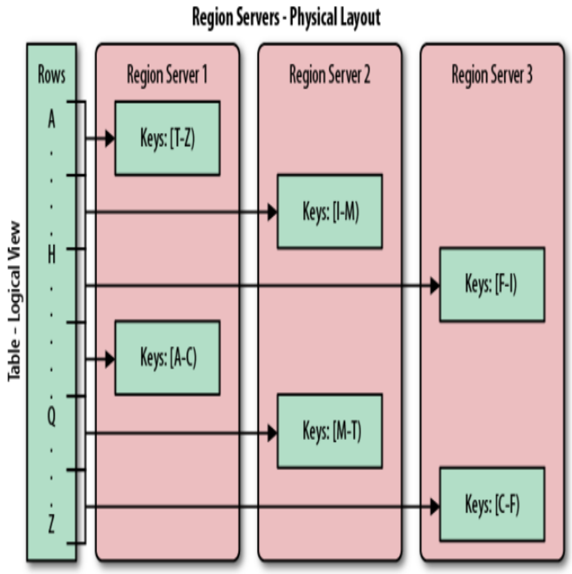

Figure 1: Rows group in regions and served by different servers [2]

gions are defined by key ranges. Regions can be automatically split by HBase or manually by defining the start key of a region. A *Region Server* manages the regions of a server. Regions are automatically split into two regions when they reach a given size or using a custom policy. Manual splitting of regions can be done at table creation time (*pre-splitting*) or later. This is advisable for instance when a hotspot is created on a region. By partitioning the region, the data can be handled by two or more servers.

Figure 1 shows an HBase deployment with three servers (pink boxes), each one hosting one region server. The keys in the table rows range from A to Z (on the left). Each region server handles two regions. Region server 1 handles keys in ranges T to Z and A to C. By default, tables have a unique region when they are created. A region is split into two regions automatically when it reaches a given limit. There are several predefined split policies, which basically split a region when the associated file reaches a given size or based on the number of regions a region server hosts. A table can be pre-split into regions either when it is created or later providing the key ranges.

Internally, the HBase Master stores metadata for instance, the location of the different regions of a table. The actual data of a region (keys and associated information) is stored in HFiles. There are as many HFiles as column families a table has. HFiles are stored in HDFS [3] to achieve high availability.

2.1 HBase Load Balancing Algorithms

HBase provides an automatic load balancer that runs on the master and distributes regions on the cluster every five minutes by default. HBase load balancer implements three algorithms [2]:

- *Simple Load Balancer.* This algorithm takes into account the number of regions each region server is managing and the load at each server. The goal is that all region servers will handle a similar number of regions by moving regions from the more loaded servers to the least loaded regions servers.

- *Favored Node Load Balancer.* This load balancing algorithm assigns favoured server for each region. The primary region server hosts the region. There are also secondary and tertiary region servers. HDFS uses the favoured servers information for creating HDFS files and placing the blocks of the file. When the primary region server crashes, the secondary takes over providing low latencies.

- *Stochastic Load Balancer.* This algorithm searches a region distribution that minimizes a cost function. This function is computed taking into account the region load, table load, data locality, MemStore sizes and HFile sizes. This algorithm has several parameters, for instance, to control the maximum number of regions to be moved, minimize the number of times the balancer will try to mutate all servers.

None of these HBase load balancers changes the region configuration. They move regions among servers to distribute the load. However, if there are several regions that become hotspots, these regions will not be split and distributed among region servers to distribute their load.

This is the goal of the Static Load Balancer we present in the next section. The load balancer distributes the keys of a table among regions in order to ensure that each region contains the same amount of keys and distributes the regions among all region servers.

3. STATIC LOAD BALANCER

The static load balancer goal is to create regions with a similar number of rows and distribute them across all regions servers in a cluster. For this purpose, the load balancer needs the table size and the key distribution. The algorithm generates a set of keys that define the new regions with the same number of keys. Then, it splits current regions according to the new regions provided by the load balancer and assigns them uniformly among region servers.

3.1 Table Histogram

In order to divide a table into regions managing a similar number of keys, the total number of rows of the table and the stored keys are needed. The table histogram scans regions reading every x number of rows (for instance, every 1000 rows). For every x rows, it stores the key of that row. For instance, if a region hosts 2500 rows and $x = 1000$, it will store three keys, 1000, 2000 and 2500. The values associated to those keys will be the keys that are stored in positions 1000, 2000 and 2500, respectively in that region. The histogram runs as an HBase coprocessor [2] so, no data is moved outside the server hosting the region.

The histogram information is used to calculate the number of rows in the table, $\#RowsTable$, the number of rows each region is currently handling, $\#RowsRegion$, the expected number of rows per region, $\#ExpectedRowsRegion$, the total number of rows that are wrongly placed, $\#WrongPlacedRows$ and the standard deviation of rows wrongly placed, $\%STDofWrongPlacedRows$. This value is used to decide

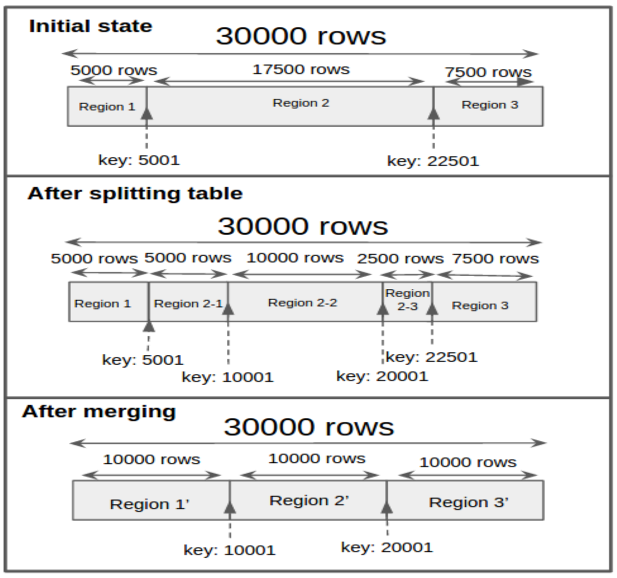

Figure 2: Load Balancer Example

whether the static load balancer should be executed. Currently, the system administrator defines a threshold, $\%STDThreshold$. If the standard deviation is greater than the threshold, the load balancer is executed.

3.2 Load Balancer

The load balancer (Algorithm 1) uses the information generated by the histogram for defining the new regions. Given the expected number of rows per region, $\#ExpectedRowsRegion$, the load balancer obtains the split points of the region by traversing the histogram. The key stored every $\#ExpectedRowsRegion$ positions will define the new regions.

For instance, Figure 2 shows a table with 30000 keys. Initially there are three regions, Region 1, Region 2 and Region 3, which handle 5000, 17500 and 75000 rows, respectively. Region 1 handles keys from 0 up to 5000, Region 2 manages the keys in the range 5001 and 22500 and so on. The final distribution the load balancer will define consists of three regions each one managing 10000 keys (each region will store a similar number of keys). If the histogram stores the keys every 10000 rows ($histogramPrecision$), the $splitPoints$ will be the keys stored at position 10001 and 20001.

Then, the algorithm splits the regions using HBase *HBaseAdmin.split()* method proving the split points. At this point the previous regions and the new ones coexists. For instance, there are 5 regions in the example in Figure 2-*After splitting table*. That is, the old regions and the new ones coexists Region 2 is split into three regions, Region 2-1, Region 2-2 and Region 2-3, with 5000, 10000 and 2500 rows each one. Only Region 2-2 will be a final region after the load balancing finishes. The other two regions will be merged with Region 1 and Region 3, respectively in order to achieve the three final regions with the same number of rows (Region 1', Region 2' and Region 3') (Figure 2-*After merging*).

As a final step in Algorithm 1 the location of the regions is stored in the Zookeeper instance running on HBase (*RegionsLocation*). This step avoids that if HBase stops and starts, by default, the regions are assigned randomly to region servers and then, the data files need to be moved to the new server where the region is handled.

Algorithm 1 Load Balancing

```
Require: $table$, $stdThreshold$
 1. $histogramPrecision = 10000$
 2. $generateHistogram(table)$
 3. $\#RS \leftarrow get\#RegionServers()$
 4. $\#Regions \leftarrow get\#Regions(table)$
 5. $\#RowsTable \leftarrow get\#RowsTable(table)$
 6. $\#ExpectedRowsRegion \leftarrow get\#ExpectedRowsRegion(table)$
 7. $\%STDofWrongPlacedRows \leftarrow getSTDofWrongPlacedRows(table)$
 8. if $\%STDofWrongPlacedRows > \%STDThreshold$ then
 9.    $splitPoints \leftarrow getNewSplitPoints(histogramPrecision, \#RowsTable, \#RS, \#ExpectedRowsRegion)$
10.    $split(splitPoints)$
11.    $merge(splitPoints)$
12.    $majorCompact(table)$
13.    $RegionsLocation(table)$
14. end if
```

4. PERFORMANCE EVALUATION

In this section we present the performance evaluation of the proposed load balancer. The evaluation has been conducted in a cluster of 11 nodes; each node is 64 core AMD Opteron 6376 @ 2.3GHz, equipped with 128GB of RAM, 1Gbit Ethernet and a direct attached SSD hard disk of 480GB running Ubuntu 12.04.5 LTS. One of the nodes is used for hosting metadata servers, HDFS NameNode, HBase Master and ZooKeeper. The rest of nodes are used as worker nodes, each one running one HDFS Data Node and four HBase-Region Servers. That is, there are 10 DataNodes and 40 RegionServers. We use the Cloudera distribution of Apache HBase with version CDH5.3.5.

The load balancer is evaluated loading the data defined by TPC-C benchmark since there are different tables with different number of columns and different number of rows. The benchmark defines 9 tables. The number of warehouses defines the sizes of the tables. In this initial evaluation, the number of warehouses is 3000. The smallest table holds 3000 rows and the largest 765 million of rows. In order to evaluate the benefits of the proposed load balancing algorithm we evaluate the performance of HBase using TPC-C with the tables split into regions with a random size. Then, we evaluate the performance of the benchmark when the regions handle a similar number of rows.

The unbalanced configuration is presented in Table 1, which shows for each table the total number of rows (*#Rows*), the number of rows of the smallest and largest regions (*#Rows Small Region and #Rows Large Region*) and the standard deviation of the rows that are wrongly placed for each table (*#STD*). For instance, warehouse and order_line tables are the smallest and largest tables, respectively.

Order_line table stores 765 million of rows. The smallest region for that table stores 11181 keys while the largest hosts 1214955735 rows (1212 millions of rows).

Table 1: Data Distribution Before Load Balancing

Table	#Rows	#Rows small region	#Rows large region	#Rows STD
warehouse	3000	6	234	63
district	30000	48	3267	795
item	100000	2	9388	22159
new_order	27M	17749	2837997	619998
orders	90M	33865	9721682	2306951
ix_orders	90M	22865	9721682	2306951
history	90M	2105753	2463901	172493
customer	90M	24592	7754305	2206718
ix_customer	90M	24592	7754305	2206821
stock	300M	18376	31182657	7463525
ix_stock	300M	113700	258520000	2509062
order_line	765M	11181	121495735	87312734
ix_order_line	765M	557433	84431120	97324041

Table 2: Data Distribution After Load Balancing

Table	#Rows	#Rows small Region	#Rows large Region	#Rows STD
warehouse	3000	75	75	0
district	30000	750	750	0
item	100000	2500	2500	0
new_order	27M	500000	684408	28020
orders	90M	1980000	2260000	743379
ix_orders	90M	1980000	2260000	43379
history	90M	2106778	2462829	172144
customer	90M	1975165	2260000	44120
ix_customer	90M	1975165	2260000	18502
stock	300M	7202284	7510000	47762
ix_stock	300M	7152000	7512000	55815
order_line	765M	18940000	19134682	29743
ix_order_line	765M	18820457	19140014	48984

4.1 Load Balancer Evaluation

In this section we present how the load balancer distributes the keys into regions given the previous distribution of data. Then, we evaluate the performance of TPC-C with both configurations and finally, we present the time for executing the load balancing algorithm.

Table 2 shows the size of the smallest region (the one hosting less keys) and the largest one for each table after running the static load balancer. The results show that the difference in number of keys hosted by these regions is less than 1%. The smallest region of table Order_line now stores 18940000 rows and the largest one stores 19134682 rows that is, 18.9 million rows and 19.1 rows respectively. We can compare those results with the ones in Table 1, which produced for the same Order_line a region with 11181 keys, while the largest region hosts 1214 millions of rows.

Table 3 shows results in terms of throughput, in transactions per minute, and latency of transactions, in milliseconds, of running TPC-C benchmark with the unbalanced and balanced regions distribution. The throughput of TPC-C with the unbalanced regions reaches 3296 transactions with an average response time of 1550.805 ms. When the regions have a similar size (i.e., after running the static load balancer), the throughput is multiplied by 10, processing 36761 transactions per minute with an average response time of 16.858 ms. That is, the response time is two orders of magnitude lower.

Finally, the execution time of the load balancer for each table of TPC-C is shown in Table 4. Table *History* is not balanced by the Static Load Balancer because it is already balanced (i.e., the

Table 3: TPC-C Execution

	Before Load Balancer	After Load Balancer
Throughput (tpmCs)	3296	36761
Avg. Latency (ms)	1550.805	16.858

Table 4: Load Balancer Execution Times

Table	#Rows	Histogram	Split Table	Merge Regions	Regions Location
warehouse	3000	00:00:02.009	00:00:06.453	00:00:11.356	00:00:10.366
district	30000	00:00:02.754	00:00:08.484	00:00:09.687	00:00:10.366
item	100000	00:00:01.946	00:00:08.070	00:00:10.022	00:00:10.400
new_order	27M	00:00:21.037	00:00:46.137	00:00:17.196	00:00:10.344
orders	90M	00:02:56.017	00:07:50.062	00:01:12.132	00:00:10.372
ix_orders	90M		00:07:29.129	00:00:44.998	00:00:10.245
history	90M	00:00:31.911			00:00:10.380
customer	90M	00:03:11.329	00:12:35.812	00:04:23.994	00:00:10.402
ix_customer	90M		00:12:41.747	00:03:11.593	00:00:10.251
stock	300M	00:10:19.886	00:39:10.252	00:07:12.418	00:00:10.312
ix_stock	300M		00:09:10.791	00:04:41.104	00:00:10.328
order_line	765M	00:37:53.463	01:59:57.023	00:11:23.552	00:00:10.421
ix_order_line	765M		00:29:52.061	00:26:18.463	00:00:10.316

$\%STDofWrongPlacedRows$ is below than 1%).

Most of the time is spent in the split process, which divides regions into several regions. Each time a region is split, a major compact process is executed in order to split the stored files (HFiles) into two. This process is very expensive for large tables (more than 100 million rows).

5. CONCLUSIONS

In this paper we have presented a Load Balancer algorithm that partitions regions into regions that manage a similar number of keys. The performance evaluation shows that this greatly improves performance. However, the execution of the load balancer is time consuming. This process should be run seldom during off-peak periods. Fault tolerance for the algorithm remains as future work.

6. ACKNOWLEDGMENTS

This research has been partially funded by the European Commission under projects CoherentPaaS, LeanBigData and CloudDBApplience (grants FP7-611068, FP7-619606 and H2020-732051), the Madrid Regional Council, FSE and FEDER, project Cloud4BigData (grant S2013TIC2894), the Ministry of Economy and Competitiveness (MINECO)under projects BigDataPaaS and CloudDB (grant TIN2013-46883 and TIN2016-80350).

7. REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

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

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[4] L. Xia, H. Chen, and H. Sun. An optimized load balance based on data popularity on hbase. In *2nd International Conference on Information Technology and Electronic Commerce (ICITEC)*, pages 234–238, Dec 2014.