HConfig: Resource Adaptive Fast Bulk Loading in HBase

Xianqiang Bao[*†], Ling Liu[†], Nong Xiao[*], Fang Liu[*], Qi Zhang[†]and Tao Zhu[†]

[*]State Key Laboratory of High Performance Computing,
National University of Defense Technology
Changsha, Hunan 410073, China
{baoxianqiang, nongxiao, liufang}@nudt.edu.cn

[†]College of Computing, Georgia Institute of Technology
Atlanta, Georgia 30332-0765, USA
{ling.liu, qzhang90, tao.zhu}@cc.gatech.edu

Abstract*—NoSQL (Not only SQL) data stores become a vital component in many big data computing platforms due to its inherent horizontal scalability. HBase is an open-source distributed NoSQL store that is widely used by many Internet enterprises to handle their big data computing applications (e.g. Facebook handles millions of messages each day with HBase). Optimizations that can enhance the performance of HBase are of paramount interests for big data applications that use HBase or Big Table like key-value stores. In this paper we study the problems inherent in misconfiguration of HBase clusters, including scenarios where the HBase default configurations can lead to poor performance. We develop *HConfig*, a semi-automated configuration manager for optimizing HBase system performance from multiple dimensions. Due to the space constraint, this paper will focus on how to improve the performance of HBase data loader using *HConfig*. Through this case study we will highlight the importance of resource adaptive and workload aware auto-configuration management and the design principles of *HConfig*. Our experiments show that the *HConfig* enhanced bulk loading can significantly improve the performance of HBase bulk loading jobs compared to the HBase default configuration, and achieve 2~3.7x speedup in throughput under different client threads while maintaining linear horizontal scalability.

Keywords—HBase; Bulk Loading; Optimization; Big Data

1. INTRODUCTION

NoSQL data stores [1] have enjoyed continued growth in many large-scale web applications over the recent years thank to their high horizontal (‘*scale-out*’) scalability. NoSQL data stores are typically key-value stores such that nothing will be shared among the key-value pairs. This enables a large dataset of key-value pairs to be partitioned into independent subsets according to keys and key ranges, which can be distributed across a cluster of servers independently. Thus, NoSQL systems can provide high throughput (a large number of *Get/Put* operations per second) through massive parallel processing. Successful examples include Bigtable [12] at Google; Dynamo [14] at Amazon; HBase [3] at Facebook and Yahoo!; Voldemort [15] at Linkedin and so forth. Among these NoSQL data stores, the open-source HBase is widely used by many Internet enterprises not only because it is used in routine operations by Facebook and Yahoo but also because HBase is an open-source implementation of a truly distributed, versioned, non-relational database modeled after Bigtable. Instead of using Google File System (GFS) [13] as the distributed storage system, HBase is developed on top of the open source Hadoop Distributed File System (HDFS) [4, 5]. *‘Facebook Messages’* [11] is a typical application at Facebook that handles millions of messages daily through HBase.

However, most of the efforts on tuning HBase performance in terms of system configuration management have been done as in-house projects. As a result, most of HBase users rely on the default configuration of HBase for their big data applications given the complexity of the configuration in terms of both the number of parameters and the complex correlation among many system parameters. Very few can answer the questions such as when will the HBase default configuration no longer be effective? What side effect should be watched when changing the default setting of a specific parameter? And how can we tune the HBase configuration to further enhance the application performance? We argue that how to setup HBase clusters with high resource utilization and high application level performance remains to be a significant challenge for system administrators, HBase developers and users.

In this paper we study the problems inherent in misconfiguration of HBase clusters, including scenarios where the HBase default configurations may lead to poor performance. For example, we will show through experiments that the default configuration may provide poor resource utilization of HBase cluster for some test cases. We will also show that some simple optimizations may even hurt HBase performance, for example, by changing the HBase Java runtime environment to bigger heapsize (from default 1GB to 4GB), the throughput performance may be degraded by 20~30% (throughput loss) compared with the default choice for some test cases. With these problems in mind, we develop *HConfig*, a semi-automated configuration manager for optimizing HBase system performance from multiple dimensions. Due to the space constraint, this paper will focus on how to improve the HBase bulk loading performance by *HConfig*. Through this case study we will highlight the importance of resource adaptive and workload aware auto-configuration management and the design principles of *HConfig*. Our experiments show that the *HConfig* enhanced bulk loading can significantly improve the performance of HBase bulk loading jobs compared to the HBase default configuration, and achieve 2~3.7x speedup in throughput under different client threads while maintaining linear horizontal scalability.

2. OVERVIEWAND PROBLEM STATEMENT

2.1 HBase Overview

HBase [3] is an open source distributed key-value store developed on top of the distributed storage system HDFS [4, 5]. An HBase system consists of four major components, as shown in Fig.1: HMaster, ZooKeeper cluster, RegionServers (RSs), and HBaseClient (HTable). HMaster is responsible for

COLLABORATECOM 2014, October 22-25, Miami, United States
Copyright © 2014 ICST
DOI 10.4108/icst.collaboratecom.2014.257304

monitoring all the RegionServer instances in the cluster, and is the interface for all metadata management. ZooKeeper [23] cluster maintains the concurrent data access to the data stored in the HBase cluster. HBaseClient is responsible for finding RegionServers that are serving the particular row (key) range. After locating the required region(s) by querying the metadata tables (.MATA. and -ROOT-), the client can directly contact the RegionServer assigned to handling that region without going through the HMaster, and issues the read or write requests. Each of the RegionServers is responsible for serving and managing those regions which are assigned to it through server side log buffer and MemStore. HBase handles basically two kinds of file types: the write-ahead log and the actual data storage through the RegionServers. The RegionServers store all the files in HDFS. HBase RegionServer and HDFS DataNode are usually deployed in the same cluster. The basic data manipulation operations referred to as CRUD (stands for Create, Read, Update, and Delete) and are implemented in HBase as Put, Get and Delete methods. The bulk loading process primarily uses the Put method. Fast bulk loading aims at distributing data to the secondary storage of the HBase cluster nodes efficiently and evenly.

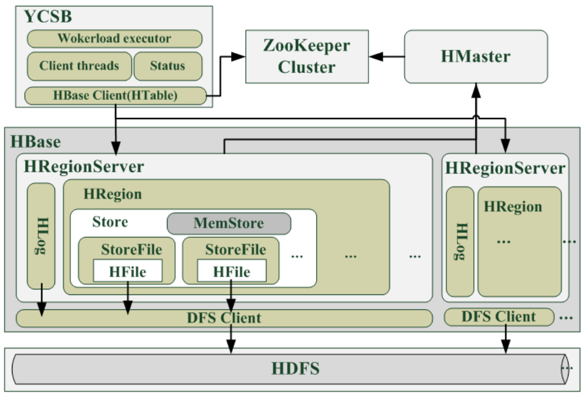


Fig. 1. HBase architecture combined with YCSB benchmark.

2.2 Bulk loading in HBase

In this section, we briefly describe the bulk loading process in HBase with default configuration. Broadly speaking, all four core components of the HBase collaborate to accomplish the bulk loading of raw datasets into the HBase cluster in three steps.

(1) HBaseClient prepares the data for bulk loading. After setting up a HBase cluster of *n* server nodes (*n*>1), we need to first load the data to the HBase store. Typically, the remote client will need to run the HBaseClient at the client side of the HBase cluster to initiate the data loading process. In order to load the raw dataset to the HBase cluster, we need to first place the key-value pairs in the write buffer of HBaseClient. The default write buffer size is usually set to 12MB. When the write buffer is full, the client sets up connections with the HBase cluster via HBaseClient in three steps: (1) the client via HBaseClient contacts the server side manager ZooKeeper [23] to get all the related regions and the region locations information based on the keys; (2) it shuffles all the buffered records according to key ranges of each region. By HBase default configuration, initially, only one region is created at a randomly selected region server (RS) when creating a new table and the intial region has the key range for any key. We can describe the initial key range as (-∞, +∞), so all the records will be routed to the same intial region initially. One parameter in the default configuratin is the threshold of when the intial region will be splited into two regions with two different key ranges as (-∞, key_1) and [key_1, +∞). When the threshold is met, the region split will be triggered by dividing the key range into two sub-ranges and each is assigned to a different region.

(2) Loading Data to the initial region on a RS. When the data records are delivered to the initial region hosted at a RS, all the records are first written into a server side buffer, and then the data will be read from the server side buffer and merged into the key range hosted in the MemStore of the intial region with lexicographic order. The default size of the MemStore is 128MB. When the MemStore is 80% full, it begins to flush records into the secondary storage of this RS managed by HDFS by creating a HFile. Each MemStore flush will create a new HFile, and one region can host several HFiles until the number of HFiles reaches the compactionThreshold (default is 3), which will trigger a minor compaction in HBase. Each minor compaction will merge the HFiles into one large HFile.

(3) Region splitting and records loading across regions and RSs. When the amount of data records loaded to a region reaches some specific threshold defined in the default configuration, the region split will be triggered. For example, the default region split policy in HBase is the *IncreasingToUpperBoundSplitPolicy,* which defines when the region split should happen:

$$(Split\ size = min(Num^3_{region/RS}*2*Flush_{size}, Max_{regionSplitSize})).$$

For example, if the raw dataset is 10GB, then the region split size for default configuration is

$\{Split_1: min(1^3*2*128MB=256MB, 10GB)=256MB,$
$Split_2: min(2^3*2*128MB=2{,}048MB, 10GB)=2048MB,$
$Split_3: min(3^3*2*128MB=6{,}912MB, 10GB)=6912MB,$
$Split_4: min(4^3*2*128MB=16{,}384MB, 10GB)=10GB, \ldots, \text{all are } 10GB\}.$

There are two types of load balancer triggers that can reassign the generated regions across the RSs: time cycle (default is 5 minutes) and the number of regions on each RS. Concretely, by setting the parameter *region.slop*, the rebalance will be triggered if the number of regions hosted by any RS has exceeded the *average+(average*slop)* regions. Upon each region split, one of the new regions will be reassigned to another randomly selected RS.

2.3 Problem Observations

Bulk loading using the default configuration suffers from a number of problems due to poor resource utilization at both HBase cluster and each region server. In this section, we present problem statement with experimental observations.

2.3.1 Experiment setup

Each node setup*:* each node of the cluster has AMD Opteron single core (Dual socket) CPU operating at 2.6GHz with 4GB RAM per core (total 8GB RAM per node), and two Western Digital WD10EALX SATA 7200rpm HDD with 1TB capacity. All nodes are connected with 1Gigabit Ethernet (125MB/sec), run Ubuntu12.04-64bit with kernel version 3.2.0, and the Java Runtime Environment with version 1.7.0_45.

HBase and HDFS cluster*:* we use HBase with version 0.96.2 and Hadoop with version 2.2.0 (including HDFS) in all the experiments. And run HBase and HDFS in the same cluster to achieve data locality (HMaster & NameNode on manager node, RegionServer & DataNode on each worker node). We use two clusters:

Cluster-small: consists of 13 nodes: 1 node hosts both HMaster and NameNode as the manager, 3 nodes host ZooKeeper cluster as the coordinators and 9 nodes host RegionServers and DataNodes as the workers.

Cluster-large: consists of 40 nodes: 1 node as manager, 3 nodes as the coordinators and 36 nodes as the workers.

YCSB benchmark:Yahoo! Cloud Serving Benchmark (YCSB) [10] is a framework for evaluating and comparing the performance of different NoSQL data stores. There are several parameters defined in this benchmark, which can be configured on the client side to generate adaptive workloads. The common parameters include the number of client threads, the target number of operations per second, the record size (the number of fields * each field size), the number of operations, the insertion order and so forth. We generate synthetic workload using YCSB load command with uniform request distribution, hash-based insert order, unlimited target number of operations per second (i.e., the YCSB client will try to do as many operations as possible). In addition, we vary the number of client threads, the record size, the number of client nodes to understand how client side configuration may impact on the bulk loading performance.

2.3.2 Unbalanced bulk loading across RegionServers

The first observation from our experiments is the unbalanced bulk loading across the cluster of RegionServers (RSs) when using the default configuration for bulk loading. Concretely, we bulk load HBase using the default configuration on the small cluster with 10 millions of data records of key-value format, which is 1KB/record and 10GB total. To gain an in-depth understanding of the problems inherent in the default configuration, we also bulk load HBase on the same cluster with 100 millions of records, a total of 100GB. Fig.2 (a) and Fig.2 (b) show the file sizes of all region servers (RSs) upon the completion of the bulk loading for 10 millions of records and 100 millions of records respectively. In the scenario of loading 10 millions of records, there are only four RSs used for handling bulk loading during the whole data loading process and other five RSs are idle with no records stored. Clearly, the default HBase configuration aims at loading data region by region and region server by region server through a conservative region split policy for data distribution. Thus, a region split will be triggered only when the data loaded to a region exceeds some default threshold. In the scenario of loading 100 millions of records to the same cluster, we observe that all 9 region servers are loaded with some portions of the input dataset but the data loading remains not well balanced across the cluster of 9 RSs (see Fig.2 (b)). To further study this result, we measured the CPU utilization for each of the four RSs that are loaded with input data for the 10 millions of records scenario shown in Fig.2 (c). In addition, we measured the throughput (#operations/sec) for both scenarios. Fig.3 (a) and Fig.3 (b) compare the throughput measurement for loading 10GB and 100GB data to the small HBase cluster respectively. Table I shows the regions and detailed data loaded on each RS for both scenarios. From Fig.3 (a), the throughput is unbalanced during the whole bulk loading process and the process can be divided into three stages. Meantime, we observe some short pauses during each of the three throughput stages, which lead to unstable throughputs even within each stage. By examining the CPU utilization trace data collected by SYSSTAT[1] [17] on the number of busy RSs, during each of the three throughput stages. We observe clearly from Fig.2 (c) that initially there is only one single busy RS (RS-1). Then during the stage 2, there are two busy RSs (RS-1 & RS-5). During the stage 3, there are four busy RSs (RS-1, RS-5, and RS-7 & RS-9).

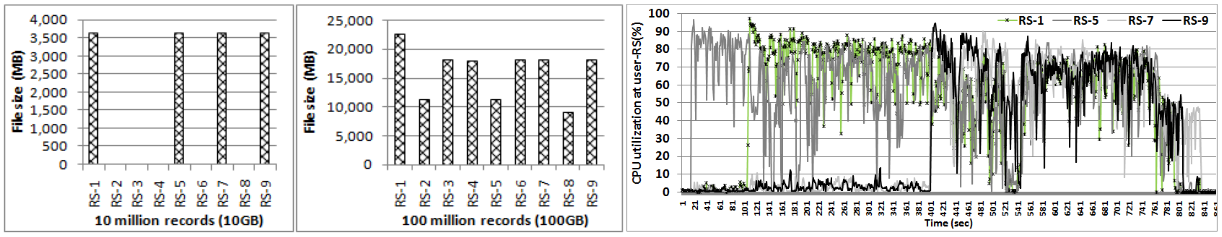
When the bulk loading dataset is increased to the 100 millions of records, we observe from Fig.3 (b) that the low throughput during the first throughput stage for the first 10 million records still exists, but the peak throughput for this scenario is much higher reaching more than 35,000 ops/sec. This shows two facts: (1) When the total size of the data for bulk loading is big enough, the generated key-range based regions will be distributed across all the RSs after the initial warming up stage and the bulk loading of 100 millions of records can be concurrently routed to all the RSs (see the region and file size details in Table I). Moreover, the average throughput of bulk loading larger 100 million records is 37% higher than the scenario of 10 millions of records, a result that is benefited from the concurrent data loading to all the RSs of the HBase cluster introduced by the incremental region splits and data loading re-distribution. .

These observations motivate us to raise a number of interesting questions: How can we improve the unbalanced bulk loading and achieve more balanced data distribution across RSs? Can we increase the bulk loading throughput to further speedup the performance of bulk loading? What configuration parameters in the default configuration of HBase should be revisited? The efforts made to answer the many questions as such motive us to develop *HConfig*, a semi-automated HBase configuration manager.

2.3.3 Inefficient resource utilization on RSs

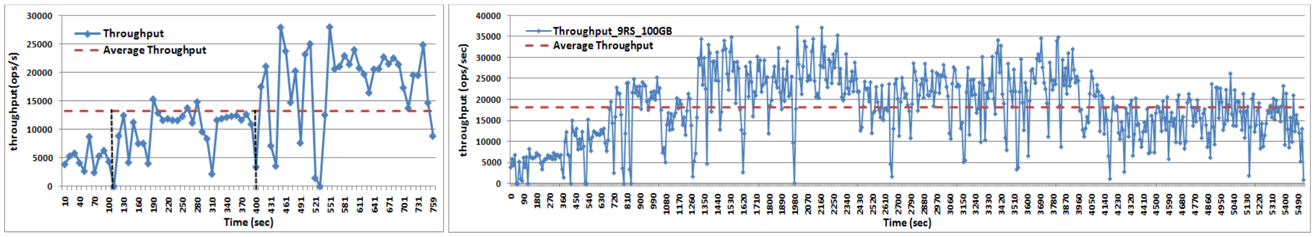
The second set of observations made from our experimental study on the bulk loading performance is the inefficient resource utilization of both cluster and individual RS nodes. First, from Fig.3 (a), the bulk loading of 10 millions of records (1KB/record) is dealing with the raw dataset of 10GB total on a HBase cluster with a total RAM capacity of all nine RSs (8GB*9=72GB RAM). However, there are only four out of nine RSs active and the average throughput of a single active RS is only about 5MB/sec, this is much less than the disk I/O bandwidth of 50~100MB/sec and network I/O bandwidth of 125MB/sec. When the bulk loading dataset is increased to 100 millions of records (about 100GB, more than the total RAM size of the cluster), we still observe the unstable throughputs in Fig.3 (b) characterized by different throughput stages and the short pauses that leads to frequent oscillation in throughputs during each stage. Although from the previous analysis, we know that one of the main causes for only a selection of RSs being active during bulk loading and for unbalanced bulk loading is the default data distribution strategy implemented through the incremental region split policy in HBase default configuration. Further understanding for root causes of short pauses that lead to throughput oscillation is as follows: we performed a series of in-depth experimental measurements by varying certain memory and disk I/O related parameters. For example, Fig.4 shows the throughput with the JVM heap size used by each RS ranging from 1GB to 4GB, the number of threads used at the client ranging from 1 to 40. Also the client threading decision has also some impact on the loading performance with 4 threads to be better than 1 thread, 2 threads, 20 or 40 threads.

[1]SYSSTAT Trace item: %user-Percentage of CPU utilization that occurred while executing at the user level (RS level)



(a) 10 million records (b) 100 million records (c) CPU trace on RSs of bulk loading 10 million records

Fig. 2. Records distributions across 9RSs with default configuration



(a) 10 million records (1KB/record) (b) 100 million records (1KB/record)

Fig. 3. Real-time throughput of bulk loading with default configurations

TABLE I. REGION AND HFILE DETAILS ON EACH RS

RS / dataset	10 million records		100 million records	
	#Region	*File Size(MB)*	*#Region*	*File Size(MB)*
RS-1	2	3,641	4	22,474
RS-2	1 (.MATA.)	0	4* (.MATA.)	11,197
RS-3	0	0	4	18,028
RS-4	0	0	4	17,941
RS-5	2	3,615	4	11,202
RS-6	0	0	4	18,078
RS-7	2	3,618	4	18,060
RS-8	1 (-ROOT-)	0	4* (-ROOT-)	8,917
RS-9	2	3,639	4	18,100

* RS-2 and RS-8 only have 3 regions for handling bulk loading

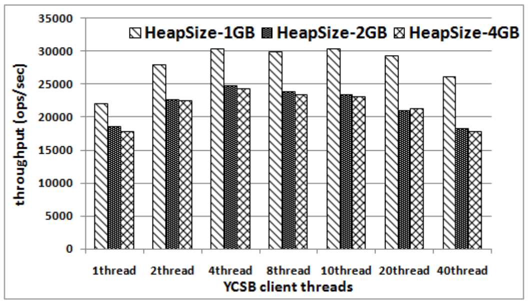


Fig. 4. Throughput of bulk loading with different heapsize

Furthermore, the frequent flushing of data from MemStore to disk (HDFS) and consequently frequent minor compactions can be caused due to inefficient utilization of the memory resources on individual RSs. The momentary decrease in throughput when the regions on some RSs are split and re-assigned to the other RSs are obviously due to the contention experienced in MemStore. These analysis results motive us to study the set of configuration parameters that can be turned automatically or semi-automatically according to cluster resource, node resource and workload characterization.

3. HCONFIG: DESIGN OVERVIEW

In this section, we first briefly discuss the objectives of *HConfig* system design and then give an overview of the set of bulk loading related parameters used in HBase configuration and analyze how these parameters may affect the bulk loading performance, followed by the concrete design considerations and optimizations implemented in *HConfig*.

3.1 Design Objectives

The *HConfig* design for bulk loading intends to help speed up the bulk loading process in HBase regardless of the workload variations and the cluster resource variations by providing the semi-automated configuration management. The concrete techniques we use focus on optimizing the utilization of both cluster resource and per-node resource at each region server. Our concrete design objectives can be summarized along three perspectives: high concurrency across all the RSs, high resource utilization on each RS and minimum bulk loading pause during the whole data loading process.

High execution concurrency. We plan to explore execution concurrency from both server and client side. At the server side, we promote the configuration of no idle RSs during the bulk loading process, enabling each RS to handle the bulk loading requests in a well-balanced manner, removing or alleviating the overload problem in the initial region hosted on single RS and during the region-split and key-range re-distribution phase. At the client side, we are interested in tunning the number of threads per client and the number of concurrent clients to better utilize the cluster resource for speeding up the bulk loading of large datasets.

High resource utilization. Instead of relying on the conservative default configuration for getting the average utiliztion of the cluster resources and per-RS resources at the best, *HConfig* by design improves the default configuration in HBase by providing resource-aware and workload adaptive configuration management, enabling HBase to run more efficiently for clusters of different size and capacity and applications with different workloads, including datasets, request types and rates and so forth, by maximizing the resource utilization and througput performance. For example, in *HConfig* we optimize the bulk loading performance through tunning the RAM and disk I/O related parameters.

Minimum load pause. By analyzing the measurement results for both data loading scenarios (10 millions and 100 millions of records) shown in Fig.2, Fig.3 and Fig.4, bulk loading pauses occur independent of the cluster size and the dataset size. We observe that those short pauses are caused primarily by three factors: a) too small MemStore and b) too frequent flushing of HFiles and consequently too many minor compactions to be performed, and c) too frequent region creation and re-distribution among RSs.

3.2 Bulk Loading Performance Related Parameters

Based on the above three optimization objectives, we select a set of configuration parameters that are highly related to the resource consumption and execution efficiency of bulk loading on both server side and client side.

3.2.1 Server-side Parameters

When bulk loading data is ready to be uploaded from the write buffer at the client, the clients can directly deliver data records to the corresponding regions hosted on different RSs of the HBase cluster. HBaseClient is the interface for the client to obtain the relevant regions for a given range of keys and the RS location information. Thus, the sever size configuration parameters are focused on RS related parameters.

TABLE II. RELATED SERVER SIDE PARAMETERS

Parameters of RS	Descriptions	Default
heapsize	The maximum amount of heap to use.	1GB
memstore. flush.size	Memstore is flushed to disk if size of the memstore exceeds this number of bytes.	128MB
memstore.block. multiplier	Block the update if memstore occupancy has reached memstore.block.multiplier * HBase.hregion.flush.size bytes.	2
memstore. upperLimit	Maximum occupancy size of all memstores in a RS before new updates are blocked and flushes are forced.	0.4
memstore. lowerLimit	Minimum occupancy of all memstores in a RS before flushes are forced.	0.38
compaction-Threshold	When the number of HFiles in any HStore (per region on a RS) exceeds this threshold, a minor compaction is triggered to merge all HFiles into one.	3
blockingStoreFiles	If more than this number of HFiles in any one HStore then updates are blocked for the Region until a compaction is completed.	10
compaction.kv. max	How many KVs to read and then write in a batch when do flush or compaction.	10
region.split.policy	determines when a region should be split. Default policy is *IncreasingToUpperBound*.	

RAM sensitive parameters. There are several configuration parameters in HBase is related directly to distributing and putting data into the secondary storage of individual RSs, including {*heapsize*, *memstore.flush.size, memstore.block.multiplier, memstore.upperLimit&lowerLimit}*. Before writing data to the secondary storage on a RS, the data needs to be placed in the MemStore of the region hosted on a RS based on their key ranges. When the MemStore overflow happens, a new HFile is created to hold the data in the MemStore and flushed to HDFS. During the flushing, all data loading to this region is blocked by the Update block until MemStore flushing is completed. The Updates block is determined by {*memstore.flush.size, memstore.block. multiplier*} or {*heapsize, memstore.upperLimit*}, and the MemStore flushing is determined by {*memstore.flush.size*} or {*heapsize, memstore. lowerLimit & upperLimit*}.

Disk I/O sensitive parameters. The following three parameters are disk I/O related configuration parameters: *compactionThreshold,blockingStoreFiles,compaction.kv.max*. The parameter {*blockingStoreFiles*} sets the threshold in terms of the number of stored HFiles and it triggers the updates block when the threshold is exceeded. The parameter {*compactionThreshold*} defines the threshold for invoking a minor compaction in terms of the number of HFiles. Too small threshold in conjunction with small MemStore size may lead to frequent minor compactions and heavy disk read/write when a compaction starts. {*compaction.kv.max*} specifies a batch disk I/O operation threshod for flusing and compaction. This parameter affects the disk read/write speed when handling the memstore flushes or compactions.

3.2.2 Client-side Parameters

In order to load the dataset from the client side to a HBase cluster, the client needs to prepare (or generate in the case of using YCSB) all the bulk loading data records and also use the HBaseClient module running at the client site to obtain the region information based on the key range and the location of RS hosting the region. Based on the region and the location of the region, the client then submits (loads) the data records to the corresponding RSs in the HBase cluster. All these operations are performed at the client side machine(s) and consume the client resources (e.g., RAM for write buffer) and network I/O bandwidth between client and the RSs of the cluster. The related parameters are listing in Table III. Among these parameters, {*WriteBufferSize, KeyValueSize*} have significant impaction the number of data records to be batch loaded to a RS. Also the client side RAM size plays a key role in configuring these two parameters. The parameters {*AutoFlush, KeysDistribution*} affect how and when the data records are distributed and loaded to the cluster of RSs. By turning off *AutoFlush*, the batch loading will be used to optimize the network I/O.

TABLE III. RELATED CLIENT SIDE PARAMETERS

YCSB Parameters	Descriptions	Default
AutoFlush	do batch writes by turning off auto flush	false
WriteBufferSize	both client side and server side use the same write-buffer size to transfer data	12(MB)
KeyValueSize	the size of a record with *N* fields and each field is *M* bytes = *N***M* (Bytes)	1KB
KeysDistribution	Key ranges generated in an ordered list by sort or hash function. The default is hash function with key ranges split uniformly by # of RSs (FNVhash64).	

3.3 Desgin Consideration and Optimizations

In this section we give an overview of the design consideration and optimizations in *HConfig*, focusing on tuning the bulk loading performance.

3.3.1 Cluster-aware optimization

This type of optimization is focused on tuning the configuration parameters that can encourage high concurrency across all the RegionServers.

PreSplit.To promote the high horizonal scalability, we design the *PreSplit* configuration, which pre-splits the big tables to be bulk loaded to the HBase cluster into independent and well-balanced reigons according the number of RSs in the cluster and distribute the regions across all the RSs based on the *KeysDistribution*. First, the cluster-aware focuses on the number of living RSs to determine how many initial regions for the target table. Then the regions are given out by splitting the table according to the *KyesDistribution*. Each region hosts a subset of the records in the input big table with certain start key and end key. This policy can avoid or minimize the regions re-assignment across RSs. One of the main challenges is to devise a well balanced data partitioning function that can partition the input data into regions according to the number of RSs. For example, keys generated by default hash function

'FNVhash64(long)' in YCSB start with the prefixes: *'user1', 'user2', 'user3', 'user4', 'user5', 'user6', 'user7', 'user8', 'user9'*. So we can split the dataset evenly into 9 regions, and initially assign one region to each RS. In *HConfig,* we allow the external data partitioning algorithms to be added through the configuration manager.

ConstantSizeRegionSplitPolicy. As a companion of the *PreSplit* policy, in *HConfig* we replace the current default policy *IncreasingToUpperBoundSplitPolicy* to *ConstantSize-RegionSplitPolicy* for region split and region re-distribution upon MemStore overflow and minor compaction. The *ConstantSizeRegionSplitPolicy* splits a region as soon as any of its store files exceeds a maximum configurable size. In comparison, the *IncreasingToUpperBoundSplitPolicy* (the default used in current HBase releases) leads to unnecessarily regions re-assignment accorss the RSs after pre-split. By replacing it in *PreSplit*, we can adaptively set the region size according to the dataset size that we can provide the optiml configuration for different datasets scenarios respectively.

3.3.2 Node-aware optimization

The node-aware optimization focuses on tuning the parameters that can impact on resource utilization at single RS, mainly centered on high efficiency in RAM and disk I/O.

RAM related tuning. The design of *PreSplit* in *HConfig* is also to obtain high utilization of the resources on each RS. One of the main design ideas is to delay the loading related update blocking and the LSM-tree [16] related minor compaction by using bigger heap (RAM) in each region. This allows to buffer more records and give priority to batch disk I/O in order to flush more records for each disk I/O. For example, the default heapsize in HBase is 1GB regardless the application tasks and the size of the memory resource on each RS. Recall the clusters used in our experiments, each RS has 8GB RAM. One simple and intuitive idea is to configure the JVM heap used by each region with a bigger heap size than 1 GB. However, simply using a bigger heapsize not only fails to deliver improved performance but also produces worse throughput than the smaller heap size. This is one of the typical misconfigurations that should be avoided. By further investigating the other RAM related configuration parameters, we discover that the MemStore related parameters should be taken into account to use the bigger heapsize efficiently. For example, we need to tune the bigger heapsize by jointly considering the settings for MemStore flush size and MemStore block multiplier, as well as the compaction threshold and the max number of HFiles hosted in a region (each flush create a HFile), to generate our optimal configuration in *HConfig*.

Concretely, MemStore is hosted in the young generation portion of the heap. Larger MemStore allows batching more records for flushing. Moreover, bigger *memstore.flush.size* delays the timing and the number of MemStore flushes by holding records in RAM longer. Thus, bigger *memstore.flush.size* also decreases the garbage collection frequency of young generation. The next complication is the setting of the parameter *memstore.block.multiplier*. Increasing the *memstore.block.multiplier* can delay the updates blocking, so when the MemStore becomes full and starts flush, if there is enough heapsize for the bulk loading, one region can utilize the multiplier MemStores to handle bulk loading records while flushing without blocking. In addition, by increasing the settings of the parameters: *memstore.upperLimit* and *memstore.lowerLimit*, the updates blocking and the MemStore flushes may be further delayed at the global level across all the RSs of the HBase cluster. But too big size may lead to runaway MemStore or long time to complete minor compaction and/or region split, possibly cause the out of memory error (OOME). Based on extensive experiments the trade-off for the heapsize and the RAM consumption, in *HConfig*, we recommend the biggest global MemStoresize to be set to no more than half of the heapsize, and the biggest *memstore.flush.size* to be 1/8 of the heapsize.

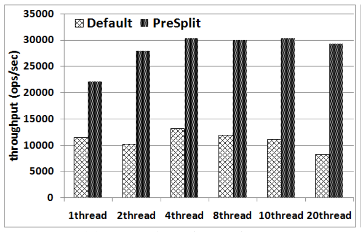
Disk I/O related tuning. Frequent flushes and frequent minor compactions can lead to higher disk I/O cost. Thus, in the bulk loading of big data, the disk I/O can easily become the bottleneck. One of the principles for optimizing the bulk loading performance is to provide high utilization of disk I/O resource during the bulk loading. For example, flushes from MemStores to HFiles stored on disk should always come first. We can increase the *compactionThreshold* to delay compactions that consume disk I/O, and increase the threshold of the *blockingStoreFiles* to delay the blocking of new updates whenever possible. Also the adaptive *compaction.kv.max* is very important to fully utilize the disk I/O bandwith, especially when the records are wide and the number of KV rows is relatively small. This enables us to use the sequential disk I/O speed for bulk loading. However, a careful trade-off is required, as too big size leads to the risk of unacceptable compaction delay or high contention in MemStore, which hurts the throughput performance of bulk loading.

3.3.3 Application-aware optimization

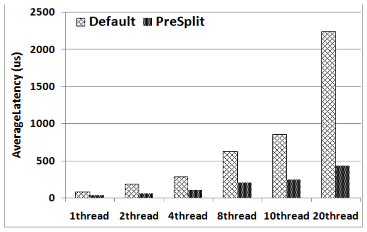
In order to further optimize the bulk loading performance based on application specific features, we can also take into account a set of client-side parameters, such as key-value record size, the application client running threads and the number of client nodes running to bulk load the data to the HBase. Our experimental results show that these application specific features also play an important role in generating the optimal configuration for further improving the throughput performance. In this paper, we focus on obtaining the optimal client side configuration by tuning the load pattern and the following parameters: *WriteBufferSize*, the number of running threads at the client, the number of concurrent client nodes, and the *KeyValueSize*. The default batch loading pattern is chosen to generate the loading workload with high client resource utilization. The number of concurrently running client nodes is determined by the number of RSs. The other parameters are determined by the resources at the client node(s) and the number of RSs, including network I/O.

4. EXPERIMENT RESULTS AND ANALYSIS

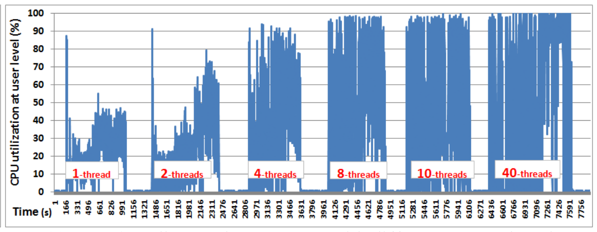
In this section, we present the details of the evaluation results for optimal configuration design compared with the default configuration used in current HBase releases.



(a)*PreSplit*: Throughput

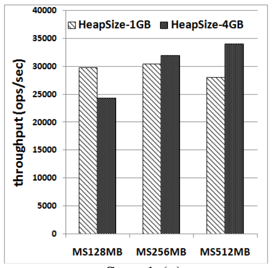


(b)*PreSplit*: Average Latency

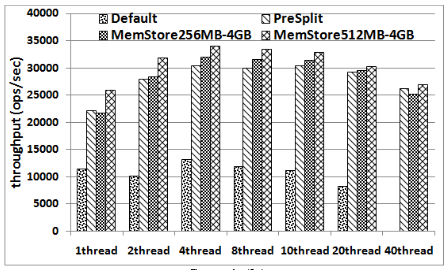


(c)*PreSplit:* Client node CPU trace with different YCSB threads

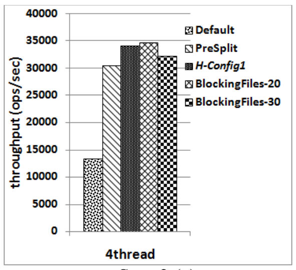
Fig. 5. Throughput and average latency of *PreSplit* configuration with different YCSB threads.



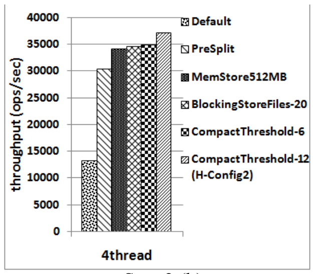
Step 1 (a)



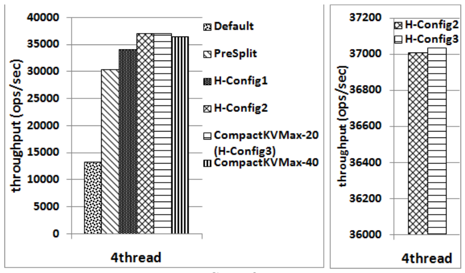
Step 1 (b)



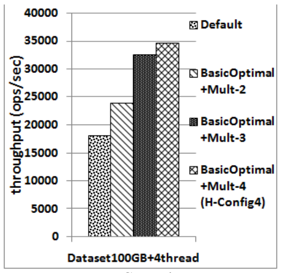
Step 2 (a)



Step 2 (b)



Step 3



Step 4

Fig. 6. Node-aware optimization throughput results.

4.1 Cluster-aware optimization evaluation

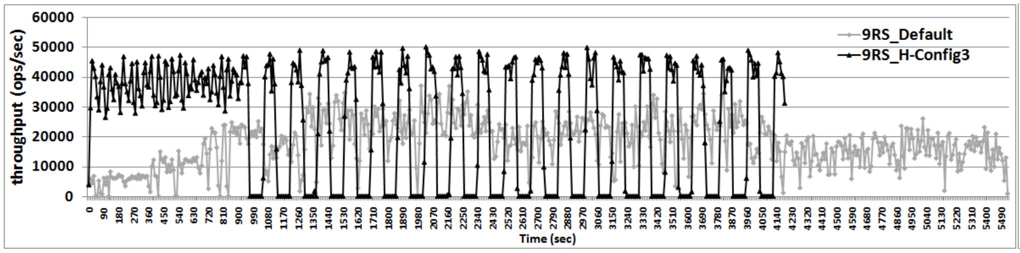
In this experiment, we use the small cluster (9RSs) to run *PreSplit* with *ConstantSizeRegionSplitPolicy* to achieve cluster-aware optimization (short for *PreSplit* configuration). According to the description of *PreSplit* design, we can pre-split the bulk loading target table 'usertable' into 9 regions as there are 9 RSs in cluster-small. From Fig.5 (a), we can see the *PreSplit* configuration significantly accelerates the throughput compared with the *Default* configuration, the speedup is from*1.9x to 3.6x* with different thread cases. And the more threads case gets the more speedup due to the high concurrency from *PreSplit*. What we should mention here is the best throughput cases of both *Default* and *PreSplit* are running with 4 client threads (*Default: 13171 ops/sec -> PreSplit: 30353 ops/sec, 2.3x speedup*), and from Fig.5. (b) the average latency of the best throughput cases are still low. Digging into the SYSSTAT CPU trace in Fig.5 (c), we find that CPU becomes the bottleneck when the client threads ≥ 8, while using less than 2 threads the CPU is underutilized. So using 4 unlimited YCSB threads will make full use of the CPU resource without network I/O bottleneck at the same time to achieve best throughput and low latency. Same situations occur in the following experiments and we use 4 threads as the optimal client thread parameter. When the key range distribution is highly skewed, a more carful configuration in terms of data partitioning is critical. In *HConfig*, we allow the external data partitioning algorithms [24] to be plugged into the data loader.

4.2 Node-aware optimization evaluation

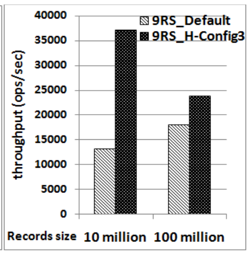
The following four steps are based on *PreSplit*:

Step 1 (memstore.flush.size). we configure the small cluster (9RSs) with default 1GB or bigger 4GB heapsize, then change the *memstore.flush.size* from default 128MB to 256MB, 512MB in this step. There are two main questions here: (1) If small heapsize with bigger *memstore.flush.size* works well (in Section II, we get performance loss when using bigger heapsize with small size) and (2) find the adaptive *memstore.flush.size* for bigger heapsize. From Fig.6. Step1 (a), the answer to question (1) is that small heapsize with bigger *memstore.flush.size* ({1GB, 512MB}) leads to performance loss, and bigger heapsize with adapative *memstore.flush.size* ({4GB, 512MB}) improves the performance and partly resolves the bigger heapsize hurts performance problem. Fig.6. Step(b) shows that the adaptive *memstore.flush.size* for bigger heapsize (4GB) is 512MB, and the improvementis 40~50% compared with the only setting 4GB bigger heapsize cases, even compared with the *PreSplit*, the improvement is 3~17% (and has *2~3.7x* speedup compared with *Default*). Also the best throughput case is running 4 client threads. So far, the optimal configuration is *$HConfig_1$ => {PreSplit + heapsize:4GB, memstore.flush.size:512MB}*.

Step 2 (blockingStoreFiles & compactionThreshold). In this step we increase *blockingStoreFiles* from default 10 to 20, 30 to delay updates blocking and then increase *compactionThreshold* from default 3 to 6, 12 to delay compaction. From the result in Fig.6. Step 2 (a), we can see the optimal blockingStoreFiles is 20, while too bigger blockingStoreFiles (e.g. 30 in this experiment) leads to throughput decrease as later blocking new updates causes MemStore too stressful in doing flushes rather than to handle new arrived records.



(a) Real-time throughput of bulk loading 100million records



(b) Different dataset size

Fig. 7. Throughput of basic optimal config*HConfig₃*.

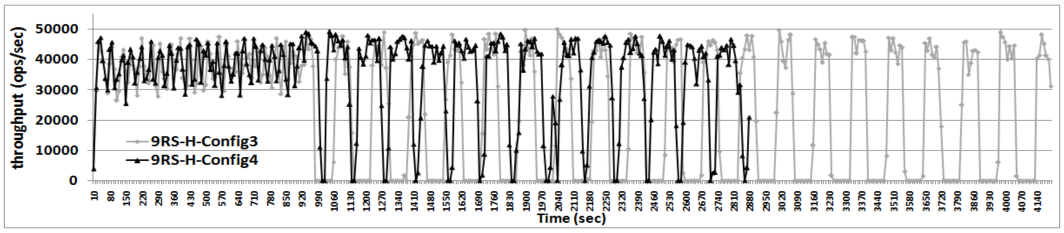


Fig. 8. Real-time throughput of bulk loading 100 million records with *HConfig4* vs. *HConfig3*.

And from Fig.6. Step 2 (b), the optiaml compactionThreshold is 12 as expect, and as too bigger compactionThreshold leads to unacceptable compaction delay risk, we just use 12 as the optimal parameter. And the best throughput case now is 37007 ops/sec (2.81x speedup compared with *Default*). So the principle about using limited disk I/O for MemStore flushes first works well. And the optimal configuration becomes *$HConfig_2$=>{ $HConfig_1$ + blockingStoreFiles:20, compactionThreshold:12}*

Step 3 (compaction.kv.max). In this step, we use the YCSB default KeyValueSize 1KB/record (100B/field*10field). And from the result in Fig.6. Step 3, there is only a tiny speedup when increasing *compaction.kv.max* from the default 10 to 20, the bigger setting with 40 even dereases the throughput. So we use 20 as the optimal parameter and the best throughput increased to 37037ops/sec (2.8102x speedup compared with Default). Till now, the optiaml configuration becomes *$HConfig_3$=>{$HConfig_2$+compaction.kv.max: 20 }.*

So far, we get the basic optimal configuration *$HConfig_3$* for loading small dataset (10 million records). However, when we use *$HConfig_3$* to load a larger dataset (100 million records), we observe some new problems. From Fig.7 (a), *$HConfig_3$* is still better than the average *Default* configuration with a much earlier finish time to bulking loading 100 million records, but periodic long loading pause occurs when finishes loading about 30 million records till the end. And from Fig.7 (b), the loading pauses leads to 36% throughput decrease (37037 ops/sec ->23781 ops/sec), and the speedup compared with *Default* is also decreased from 2.81x to 1.32x. As the disk I/O resources are already full used in *$HConfig_3$*, so we turn to increase the whole RAM used by MemStores to get the optimal configuration for bulk loading very large dataset.

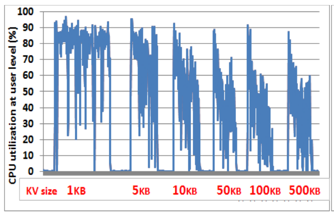
Step 4 (memstore.block.multipiler). When dataset becomes much larger, although compaction is delayed in basic optimal configuration, but it should not be delayed too much to cause unacceptable compaction delay risk. And when doing the compaction and the periodic MemStore flushing at the same time, new updates blocking occurs due to shortage of disk I/O as well as too stressful MemStore also shows as periodic pauses. In this step, we increase the global MemStores of one region to further delay updates blocking based on *$HConfig_3$* by using bigger memstore.block. multipiler (changed from Default 2 to 3, 4). From the result in Fig.6. Step 4, for the best throughput case (multipiler is 4), there is a 46% throughput improvement compared with *$HConfig_3$* and 1.92x speedup compared with Default. Then from Fig.8, when the loading records dataset is less than about 30 million, bigger memstore.block.multipiler behaves as the same as the default samller one. But bigger multipiler can significantly shorten the loading pauses due to further delay updates blocking caused by too stressful MemStores. And the optimal configuration is

$HConfig_4$=>{ $HConfig_3$ + memstore.block.multipiler:4 }.

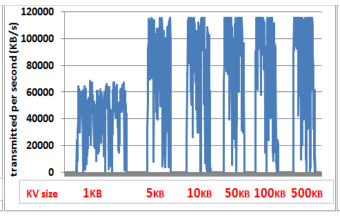
4.3 Application-aware optimization evaluation

In the above cluster-aware optimization experiment, we already get the optimal running threads of YCSB benchmark on our setup client node. In this experiment, as bulking loading requests distribution is uniform, so the main feature of the benchmark affects performance much is the KeyValueSize (KV size) and we change the KeyValueSize from 1KB to 5KB, 10KB, 50KB, 100KB, 500KB to get the client size optimal configuration (each record has 10 fields, and we use the default WriteBufferSize 12MB in YCSB). As all the records are real-time generated by client threads, so we analysis the client node resources and network I/O utilization first. From the trace results in Fig.9 (a) and (b), when the KV size is default 1KB, the CPU becomes the bottleneck and the network I/O utilization is around half of the GbE (1Gb Ethernet). And when KV size is 5KB or bigger, the GbE becomes the bottleneck instead of the CPU of client node. So when the records generated by the application hosted on HBase are always $\leq$ 5KB, the bulk loading (batch model) is more CPU sensitive than network I/O and bigger WriteBufferSize should be configured to make full use of the network I/O. And when the records are > 5KB, better network I/O improves the loading performance. So we should increase the WriteBufferSize to 2~3x to get the optimal configuration.

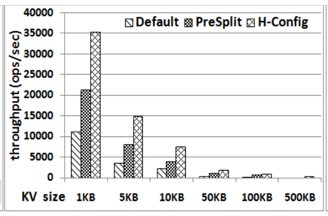
Moreover, from Fig.9 (c) and (d), our *HConfig* works well from 1KB to 500KB cases as maintaining with *3~4x* speedups. And the 5KB KV size case gets the highest speedup due to both full utilization of network I/O and CPU, it also verifies that we can still optimize bulk loading performance by improving network I/O utilization with bigger WriteBufferSize based on the general optimal configuration *$HConfig_4$* of the server side.



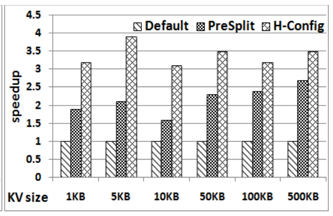
(a) CPU trace



(b) Network I/O trace

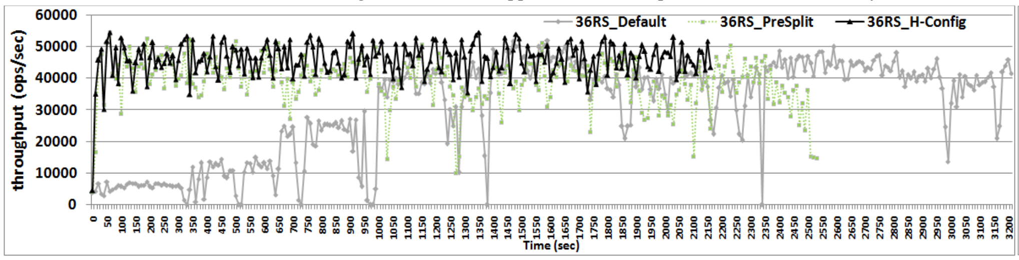


(c) Throughput (ops/sec)

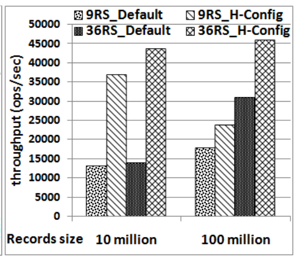


(d) Speedup

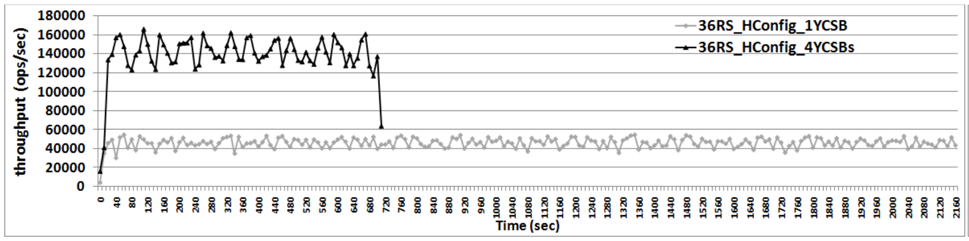
Fig. 9. Results for application-aware optimization with varied key-value size.



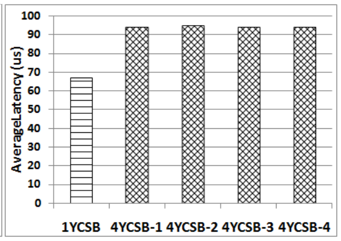
(a) Real-time throughput on cluster-large with different configurations



(b) Throughput



(c) Real-time throughput on cluster-large with single and multiple client nodes



(d) Average Latency

Fig. 10. Results for scale-out evaluation.

4.4 Scale-out evaluation

We use the large cluster (36RS) and multiple client nodes (4YCSB running nodes). From Fig.10 (a), when we use the optimal configuration *$HConfig_4$* load 100 million records, the periodic long pause is gone. Because records are well balanced loading to 36RSs and the pressure of RAM or disk I/O for single RS is much less than loading 100 GB records to 9RSs, and this indicates well scalability here. From Fig.10 (b), when the loading records dataset is small, both small and large cluster have significant speedup with optimal configuration. While when the records dataset becomes much larger, the *Default* on large cluster (*36RS_Default*) performs much better than *Default* on small cluster (*9RS_Default*) with 1.72x speedup, even better than the optimal configuration on small cluster (*9RS_HConfig*). This indicates that the *Default* average configuration for HBase is fit for big data process on large data centre environment, while the optimal configuration on large cluster (*36RS_HConfig*) still gets about 50% throughput improvement compared with *36RS_Default*. However, from the results in Fig.10 (a) and (b), there are also two questions: (1) The throughput of optimal configuration performs only the same level as *PreSplit* as well as the last stage of *Default*, that means larger heapsize and other optimal parameters have no help to the bulk loading performance. (2) Although optimal configuration on large cluster gets throughput improvement, but the speed up is much smaller than small cluster under the similar pressure on RAM and disk I/O for each RS (The speedup for *36RS_HConfig : 36RS_Default* is 1.5x, while *9RS_HConfig : 9RS_Default* is 2.8x). Both of the above problems due to the bulk loading records generated by one single client node can't feed the 36RSs up to push 36RSs running at full capacity. So we use multiple 4 client nodes to generate enough records concurrently to feed the 36RSs up. To be more specific, each client node runs 4 YCSB client nodes with 4 threads per node (so the 36RSs handle 16 client threads concurrently), and the 100 million records set is divided into 4 subsets for each client node to load a subset including 25 million records. From the throughput results in Fig.10. (c), when using 4 YCSB client nodes, the throughput for 4 multiple YCSB nodes is the superposition of each node's throughput and the speedup is *3x* (3x=0.75x*4) compared with single client node. Although the throughput of each node among multiple client nodes is decreased (while the average latency of each node is still acceptable, see Fig.10 (d)), but the whole throughput achieves considerable speedup and almost has linear scalability with our optimal configuration. Moreover, as using 16 threads on single client node faces CPU bottleneck (as Fig.5 (c)), we recommend multiple client nodes with optimal threads to bulk load large cluster.

5. RELATED WORK

We present the recent related research works mainly in the evaluation of NoSQL data stores and the optimization for big data process on HBase as following:

Cooper et al. [10] present YCSB framework to compare the performance of new generation of NoSQL data stores and report results for HBase [3], Cassandra [18], PNUTS [19], and a simple shared MySQL implementation based on their defined synthesis workloads. Patil et. al [20] extend YCSB and build YCSB++ to support advanced features for more complex evaluation of NoSQL systems, such as multi-tester coordination, eventual consistency test, doing anticipatory configuration optimization and so on. These benchmark tools are targeted at a wide range of systems and focusing on benchmark development, both YCSB and YCSB++ turn to use the default average configuration to evaluate the target systems, and YCSB does not optimize the configuration of underlying target systems. Although YCSB++ does optimization such as B-tree pre-splitting or bulk loading in benchmark tests, but benchmark-level optimization should take the wide range of target systems into account to be fair to each one, and always is general optimization. Our optimization solution *HConfig* takes application-level (benchmark-level) as well as cluster-level, node-level resources into account to generate the optimal configuration for HBase.

Harter et. al [2] present a detailed study of the Facebook Message stack as the case study to analysis HDFS under HBase, and suggest to add a small flash layer between RAM and disk to get best performance improvement with certain budget based on the cost simulations. Huang et. al [21] use high performance networks to optimize HBase read/write performance by exploiting the Infiniband RDMA capability to match the object delivery model in HBase. These works can significantly improve HBase performance by taking advantage of novel data storage and transfer technologies.

Das et. al [9] present G-Store implemented based on HBase to provide efficient, scalable, and transactional multi key access with low overhead. Similarly, Nishimura et. al [22] proposed MD-HBase to extend HBase to support advanced features. These functionality optimizations are orthogonal to our work, and our optimal configuration can improve the performance with significant speedup for any HBase applications.

6. CONCLUSION AND FUTURE WORK

We have presented *HConfig,* a semi-automated configuration management system. We show through our experimental study that the default configuration in current HBase releases can hurt the average performance in bulk loading regardless of the dataset sizes and the cluster sizes. The problems inherent in misconfiguration are addressed by *HConfig* with providing resource adaptive and workload aware configuration management. Our experiments show that the *HConfig* enhanced bulk loading can significantly improve the performance of HBase bulk loading jobs compared to the default configuration, and achieve 2~3.7x speedup in throughput under different client threads while maintaining linear horizontal scalability.

ACKNOWLEDGEMENT

This work is carried out when Xianqiang Bao is a visiting PhD student in Georgia Institute of Technology supported by scholarship from the China Scholarship Council. Authors from Georgia Tech are partially supported by NSF under Grants IIS-0905493, CNS-1115375, IIP-1230740 and a grant from Intel ISTC on Cloud Computing. Authors from NUDT are partially supported by the National High Technology Research and Development 863 Program of China under No.2013AA013201, and the NSF of China under Grant Nos. 61433019, 61232003, 61025009, 61170288, 61332003, and 61120106005.

REFERENCES

[1] R. Cattell, “Scalable SQL and NoSQL Data Stores,” Proceedings of ACMSIGMOD’10 Record, vol. 39, No.4, pp. 12–27, 2010.

[2] T. Harter, D. Borthakur, S. Dong, A. Aiyer, et al."Analysis of HDFS Under HBase: A Facebook Messages Case Study". Proceedings of USENIX FAST'14, Santa Clara, CA, February 2014

[3] Apache HBase, http://HBase.apache.org/

[4] Apache Hadoop, http://hadoop.apache.org/

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. Proceedings of IEEE MSST'10, Incline Village, Nevada, May 2010.

[6] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash,et al. "Apache Hadoop Goes Realtime at Facebook". Proceedings of ACM SIGMOD'11, Athens, Greece, June 2011.

[7] K. Lee, L. Liu. "Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud", Proceedings of SC'13, Denver, CO, USA, November 17-21, 2013.

[8] B. Wu, P. Yuang, H. Jin and L. Liu. “SemStore: A Semantic-Preserving Distributed RDF Triple Store”, Proceedings of ACM CIKM'14. Nov. 3-7, 2014

[9] S.Das, D. Agrawal, A. Abbadi. "G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud",Proceedings of ACM SoCC’10, Indianapolis, Indiana, June 10-11 2010.

[10] B.F.Cooper, A.Silberstein, E. Tam, R. Ramakrishnan,and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” Proceedings of the ACM SoCC’10, Indianapolis, Indiana, June 10-11 2010.

[11] K. Muthukkaruppan,“Storage Infrastructure Behind Facebook Messages.” Proceedings of HPTS’11, Pacific Grove, California, October 2011.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al, “Bigtable: A Distributed Storage System for Structured Data,” Proceedings of USENIX OSDI’06, WA, November 2006.

[13] S. Ghemawat, H. Gobioff, and S. Leung, “The Google File System,” Proceedings of ACM SOSP’03, NY, USA, October 19-22 2003.

[14] G.DeCandia, Hastorun, D., Jampani, M., Kakulapati, G., et al. "Dynamo: Amazon’s highly available key-value store". Proceedings of ACM SOSP’07, pp. 205–220, Stevenson, WA, 2007

[15] Voldemort, http://www.project-voldemort.com/voldemort/

[16] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The log-structured merge-tree (LSM-tree)". Acta Informatica, 33(4):351-385, 1996.

[17] SYSSTAT, http://sebastien.godard.pagesperso-orange.fr/

[18] A. Lakshman, P. Malik, and K. Ranganathan. "Cassandra: A structured storage system on a P2P network". Proceedings of ACMSIGMOD’08,Vancouver, Canada, June 9-12, 2008.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, et al, “PNUTS: Yahoo!'s hosted data serving platform”, Proceedings of the VLDB Endowment, v.1 n.2, August 2008

[20] S. Patil, M. Polte, K. Ren, W. Tantisiriroj,et al, “YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores,” Proceedings of the ACM SoCC’11.

[21] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur-Rahman, et al. "High-Performance Design of HBase with RDMA over InfiniBand", Proceedings of IEEE IPDPS'12.

[22] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services",Proceedings of IEEE MDM'11, Vol.1 pp 7-16.

[23] Apache ZooKeeper™. http://zookeeper.apache.org/

[24] Kisung Lee and Ling Liu. "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning", VLDB 2014, Volume 7.