Policy-driven Configuration Management for NoSQL

Xianqiang Bao[*†], Ling Liu[†], Nong Xiao[*], Yang Zhou[†], Qi Zhang[†]

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

[†]Distributed Data Intensive Systems Lab, School of Computer Science,
Georgia Institute of Technology, Atlanta, Georgia 30332-0765, USA
lingliu@cc.gatech.edu, {yzhou, qzhang90}@gatech.edu

***Abstract*—NoSQL systems have become the vital components to deliver big data services in the Cloud. However, existing NoSQL systems rely on experienced administrators to configure and tune the wide range of configurable parameters in order to achieve high performance. In this paper, we present a policy-driven configuration management system for NoSQL systems, called PCM. PCM can identify workload sensitive configuration parameters and capture the tuned parameters for different workloads as configuration policies. PCM also can be used to analyze the range of configuration parameters that may impact on the runtime performance of NoSQL systems in terms of read and write workloads. The configuration optimization recommended by PCM can enable NoSQL systems such as HBase to run much more efficiently than the default settings for both individual worker node and entire cluster in the Cloud. Our experimental results show that HBase under the PCM configuration outperforms the default configuration and some simple configurations on a range of workloads with offering significantly higher throughput**.

I. INTRODUCTION

NoSQL systems have become the vital components for many scale-out enterprises [3, 14, 15], due to their schema free design and shared nothing key-value abstractions to support many science and engineering applications as well as real-time web services [4, 16, 23]. However, existing NoSQL systems rely on experienced system administrators to configure and tune the wide range of system parameters in order to achieve high performance. Furthermore, most performance tuning efforts for NoSQL systems are done as in-house projects. As a result, a majority of NoSQL users just use the default configuration for their cloud applications. And it is the much more daunting challenge for developers and users with average experience on NoSQL to be truly familiar with a large set of parameters and understand how they should interact to bring out the optimal performance of a NoSQL system for different types of workloads. For example, very few can answer some of the most frequently asked configuration questions: When will the default configuration no longer be effective? What side effect should one watch out for when changing the default setting of specific parameters? Which configuration parameters can be tuned to speed up the runtime performance of the write-intensive applications? With the increased popularity of NoSQL systems, the problem of how to setup NoSQL clusters to provide good load balance, high execution concurrency and high resource utilization becomes an important challenge for NoSQL system administrators, developers and users as well as cloud service providers [17].

In this paper, we present a workload aware and policy-driven configuration management system for NoSQL systems, called PCM. First, we argue that it is essential to understand how different settings of parameters may influence the runtime performance of NoSQL system under different NoSQL workloads. We use PCM to identify workload sensitive configurable parameters and capture the tuned parameters for a classification of workloads as configuration policies. Second, we use PCM to analyze the impact of a range configuration parameters and their interactions on the runtime performance of NoSQL systems in terms of read and write workloads. We show that simply changing some parameters from their default settings may not bring out the optimal performance, and the tuned parameter settings for one type of workloads such as bulk loading, may not be effective for another type of workloads, such as read-intensive workloads. Last but not least, we show that the adaptability for various workloads and the tuned configurations recommended by PCM can enable NoSQL systems such as HBase, to run much more efficiently than the default settings for both individual worker node and the entire cluster in the cloud. We evaluate PCM extensively using two scales of HBase clusters on a set of representative workloads. The experimental results show that the typical NoSQL system HBase powered by the PCM configurations significantly outperforms the default configuration on a range of workloads with different dataset sizes, offering significantly higher throughput while maintaining high adaptability to various workloads and almost linear scalability.

II. POLICY-DRIVEN CONFIGURATION: AN OVERVIEW

A. Characterization of NoSQL Workloads

In PCM, we characterize NoSQL workloads into three major categories according to the read/write ratio of client requests: *write-intensive*, *read-intensive*, and *read/write-mix*. Then we further characterize write-intensive workloads into three sub-categories: *bulk loading* (BL), *write-only* (WO) and *write-mostly* (WM). Similarly, we characterize read-intensive workloads into two sub-categories: *read-only* (RO) and *read-mostly* (RM); and read/write-mix workloads into three sub-categories: read/write-mix with similar read/write proportion (MixRW), read/write-mix with more read proportion (MixR) or with more write proportion (MixW). In this work, workloads are generated by four baseline data manipulation operations of the workload generator YCSB [5]: *Insert, Update, Read* and *Scan* (more details in Table I).

1) Write-intensive workloads: Bulk loading (BL) loads the prepared dataset to an empty target database. BL requests are implemented as Insert operations in YCSB. Write-only (WO) workloads are further characterized into WO-Insert (InsertProportion=100%) and WO-Update (UpdateProportion =100%). Write-mostly (WM) workloads refer to the workloads with a small amount of read workloads (less than 10%) added into the WO workloads.

TABLE I. MAIN OPERATIONS DEFINED IN YCSB

Type	Description	YCSB Parameter
Insert	Insert a new Key-Value (KV) record[a].	InsertProportion
Update	Update a KV record by replacing the value of one filed.	UpdateProportion
Read	Read a record to get the value of either one randomly chosen field or all fields.	ReadProportion
Scan	Scan from a randomly chosen start key and fetch back randomly chosen number[b] of KV records in Keys' order.	ScanProportion

[a] Each KV record has a primary string key as Key and a number of string fields as Value.
[b] The number of KVs to scan for each operation is randomly chosen between (1, MaxScanLength)

2) Read-intensive workloads: RO-Read workloads are those with ReadProportion=100% for point based reads. RO-Scan denotes the range read with ScanProportion=100%. Read-mostly (RM) workloads refer to those with a small amount of writes (less than 10%) added into the RO workloads.

3) Read/write-mix workloads: Read/write-mix (MixRM) workloads have similar read/write proportions, e.g., around 40%~60% for both read and write. Read/write-mix with more read (MixR) workloads have much more read proportion, around 60%~90%. Read/write-mix with more write (MixW) workloads have much more write proportion, e.g., 60%~90%.

B. HBase Overview

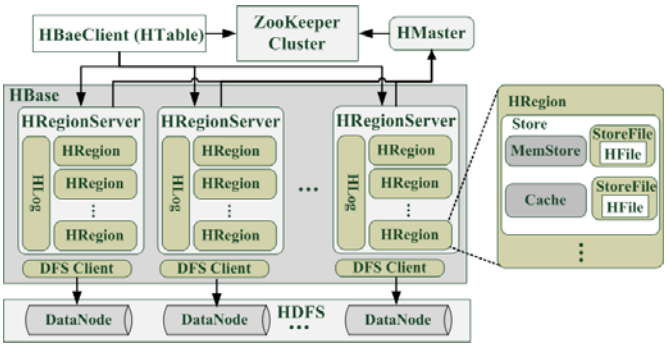


Fig. 1. HBase Architecture Overview.

TABLE II. RELATED SERVER SIDE PARAMETERS

Scope	Parameters	Descriptions
Cluster	region.split.policy	To determine when a region to be split.
RS	heapsize	The maximum amount of heap to use.
	memstore. upperLimit	Maximum occupancy size of all memstores in a RS before new updates are blocked and flushes are forced.
	memstore. lowerLimit	Minimum occupancy of all memstores in a RS before flushes are forced.
	handler.count	Count of RPC Listener instances spun up on RegionServers
Region	memstore. flush.size	Memstore is flushed to disk if size of the memstore exceeds this number of bytes.
	memstore. block.multiplier	Block updates if memstore occupancy has reached memstore.block.multiplier *memstore.flush.size bytes.
	block.cache.size	Percentage of maximum heap to allocate to block cache used by HFiles.
	max.filesize	Maximum HStoreFile size.
Store (HFile)	compaction Threshold	When the #HFiles in any HStore exceeds this, a minor compaction is triggered to merge all HFiles into one.
	blockingStoreFiles	If more than this #HFiles in any one HStore then updates are blocked for the Region until a compaction is completed.
	compaction.kv.max	How many KVs to read and then write in a batch when do flush or compaction.

HBase [2, 18] is an open source distributed key-value store developed on top of the Hadoop distributed file system HDFS [8, 9]. It consists of four major components (see Fig.1): HMaster, ZooKeepercluster, RegionServers (RSs), and HBaseClient (HTable). HMaster is responsible for monitoring all RS instances in the cluster, and is the interface for all metadata management. ZooKeeper [19] cluster maintains the concurrent access to the data stored in the HBase cluster. HBaseClient is responsible for finding the RSs that are serving the particular row (key) range called a *region*. After locating the required region(s) by querying the metadata tables (.MATA. and -ROOT-), the client can directly contact the corresponding RegionServer to issue read or write requests over that region without going through the HMaster. Each RS is responsible for serving and managing the regions those are assigned to it through server side log buffer, MemStore and block cache. HBase supports two file types through the RegionServers: the write-ahead log and the actual data storage (*HFile*). The RSs store all the files in HDFS. Table II shows a set of parameters related to performance tuning for RSs and regions. Specifically, *region.split.policy* is an important parameter to determine the data layout across all RSs, which has significant impact on load balance. HBase currently has four split policies available for configuration:

- *IncreasingToUpperBound* split policy, the default policy for HBase version 0.94 and later, which triggers region splits when region size meets the following threshold:

$$(Split\ point = min(Num^3_{region/RS} * 2 * MemStore_{flushSize}, Max_{regionFileSize})$$

For example, if *memstore.flush.size* is 128MB and max.filesize is 10GB, the region split process is carried out as follows:

{Initial: new created table allocates only one region by default,
$Split_1$*:min(*1^3**2*128MB=256MB,10GB)=256MB,* $Split_1$ *point;*
$Split_2$*:min(*2^3**2*128MB=2,048MB,10GB)=2048MB,* $Split_2$ *point;*
$Split_3$*:min(*3^3**2*128MB=6,912MB,10GB)=6912MB,* $Split_3$ *point;*
$Split_4$*:min(*4^3**2*128MB=16,384MB,10GB)=10GB,…, all are 10GB.}*

- *ConstantSize* split policy, which triggers region splits when the total data size of one Store in the region exceeds the configured *max.filesize*.
- *KeyPrefix* split policy, which groups the target row keys with configured length of prefix such that rows with the same key prefix are always assigned to the same region.
- *Disabled* split policy, which disables the auto-split processing such that region splits only happen by manual split operations of the system administrators.

C. Motivation and Design Guideline

We first motivate the design of PCM through an illustrative example, to show the performance gain by identifying critical set of parameters and their configuration optimization recommendations. Specifically, we want to bulk load 10 million records of 1KB each into an empty table on a small HBase cluster with 9RSs (see Section 5 for more details). Table III shows the main steps for a user to get a bulk load configuration that offers better performance than the default policy in HBase. From steps S.2 and S.3, we see that simply using a larger *heapsize* does not always guarantee performance improvement for bulk loading workloads. We observe from S.4 to S.6 that replacing the default load balancing strategy by *pre-split* policy, we can significantly improve the bulk loading throughput (S.4). However, by combining with larger heapsize, the throughput drops compared to S.4 (S.5 and S.6). This is

because simply enlarging the heap size does not help the throughput at all. Instead, if we also increase the *memstore.flush.size* in addition to heap size, we can achieve higher speedup in throughput (S.7 and S.8). If we further tune the storage related parameters, e.g., *blockingStoreFiles* and *compactionThreshold*, we can further improve the performance of bulk loading due to high resource utilization of memory and disk I/O (S.9).

TABLE III. POLICY EXAMPLE FOR BULK LOAD OF HBASE

Step	Configuration[a]	Speedup	Remark
S.1	Default	1x	Set throughput of default as baseline
S.2	S.1+4GB Heap	1.0x	Just use bigger heap with other parameters default is useless.
S.3	S.1+6GB Heap	1.0x	
S.4	S.1+PreSplit	2.7x	PreSplit leads to load balance
S.5	S.4+4GB Heap	2.2x	Bigger heap still inefficiency used and hurt PreSplit throughput.
S.6	S.4+6GB Heap	2.1x	
S.7	S.6+256MB MS	2.8x	MS is memstore size, bigger heap should configure bigger memstore.
S.8	S.6.+512MB MS	3.0x	
S.9	S.8+20BF&12CT	3.3x	Tuning storage related parameters can achieve further improvement.

[a] More details about the related parameters tuning in Section III & IV.

Through this motivating example, we make two arguments:

1) One (default) configuration cannot fit all: Although the default configuration may be a good choice for average performance, our example shows that opportunities exist to further optimize NoSQL system performance by identifying and tuning the workload related parameters. For the example bulk loading workload, we can provide an over *3x* speedup for throughput performance by tuning configuration. It is also known that parameters tuned for performance of write intensive workloads hardly ever work well for performance tuning of read intensive workloads. Thus, we need an extensible and customizable policy-driven autonomic configuration system such that it can switch the policies according to the workload changes, and it allows administrators, developers and users to add their own performance tuning policies equipped with trigger and adaptation conditions.

2) Interactions of the critical parameters and tuning policies can be very complex: Most of the popular NoSQL systems, including HBase, have a large number of configurable parameters, each with different scope (see Table II). Also, the same parameters may have different impacts for different workloads. The interaction between the parameters can be very complex. The common relationships include dependency based correlation and competition based correlation, such as those shown in Table III (S.2/3/5/6). Furthermore, NoSQL systems are known for their elasticity by running on different cluster setups. Thus, any parameter tuning strategies for configuration management should be transparent to the cluster setups and should maintain the horizontal scalability of NoSQL [16].

III. WORKLOAD ADAPTIVE PCM

A. Design Objectives

The main objectives for PCM design are three folds: (1) The configuration policy and parameter tuning recommended by PCM should provide high performance speedup compared to default or baseline configuration policy. (2) The PCM should provide automated or semi-automated configuration optimization adaptive to a selection of NoSQL workloads at both the cluster level and the compute node level. (3) The PCM should be light weight, extensible and customizable, allowing easy plug-in of new configuration policies for new types of workloads and seamless upgrade of existing configuration to improve the runtime performance of the NoSQL system.

Consider write-intensive, read-intensive and read/write mixed workloads, for HBase, write-intensive workloads depend on parameters such as *heapsize*, memstore related parameters, such as *memstore flushing*, *write blocking*, hfile related parameters such as *hfile compaction*. Adequate parameter configuration can have significant impact on HBase write operation behavior. In contrast, read-intensive workloads depend on some different parameters or different settings of the same parameters, such as *block cache* in *heapsize*, hfile *block size*. Because these parameters can have significant impact on cache hit ratio. The read/write mixed workloads depend on both write-intensive and read-intensive workloads. Thus, the read/write proportion can help to determine how to tune the competitive parameters between read and write, such as how to configure the heap proportion to memstore for write tuning and cache for read tuning.

B. Two Level Configuration Tuning Strategies

NoSQL systems are designed to run on a cluster of nodes. We argue that the configuration optimization for NoSQL should be cluster-aware and node-aware.

Cluster-aware tuning strategies should focus on tuning the configurable parameters which can improve the overall performance of the cluster. For HBase, we first identify a set of parameters those can be tuned to improve concurrency and load balance across the RSs. For example, the *PreSplit* strategy is designed to pre-split the input table into independent and well balanced regions according to the number of RSs in the cluster and distribute the data across the RSs based on the keys distribution. In addition, we need to improve concurrent execution at each RS through multiple regions, we pre-split the large input dataset into P number of regions, $P = N$ times #RSs. So that each RS will have N regions. During bulk loading, we use *PreSplit* with *ConstantSize* split policy to reduce the high cost of both region splits and re-assignment cost occurred in default configuration. However, after bulk loading for write-most or mixed read/write workloads, we use the *IncreasingToUpperBound* split policy to further split the regions when the *max.filesize* exceeds its threshold. And this enables high concurrency across RSs.

Node-aware tuning strategies should be centered on tuning the parameters related to per-node resource utilization to improve the runtime performance of individual server node. For the bulk loading (as well as write-intensive) workloads, we can delay the update blocking and the LSM-tree [13] related minor compaction. In order to perform memory related tuning, we use adaptive heap size in each RS (around 1/2~3/4 of the total memory size), which allows us to buffer more records and give priority to batch disk I/Os in order to flush more records for each disk I/O. For HBase, the following four are the most important *memstore* related parameters: *upperLimit*, *lowerLimit, flush.size, block.multiplier,* to achieve more efficient use of the bigger heap per-RS. Similarly, for disk I/O related tuning, frequent flushes and minor compactions can lead to higher disk I/O cost. One way is to let the disk I/O utilization for flushes from MemStores to HFiles

stored on disk always come first by increasing the *compactionThreshold* to delay compactions that consume disk I/O much, and increase the threshold of the *blockingStoreFiles* to delay the blocking of new updates whenever possible. However, a careful trade-off is required here, as too big *compactionThreshold* and *blockingStoreFiles* may lead to unacceptable compaction delay, high *memstore* contention.

In contrast, for the read-intensive workloads, the tuning strategy focuses on the cache hit ratio, which has significant influence on read performance. For RAM related tuning, we can increase the *heapsize* and *block.cache.size* to allow read-intensive workloads to load more records into heap after all the meta data (such as index and bloom filter data) has been loaded into memory. For disk I/O related tuning, the parameter hfile *block size* is very important. A smaller block size is more efficient for point read workload and a bigger block size is better for range read workload. Also adequate configuration of major compactions can be beneficial, especially for range reads. Next, to design the read/write mixed tuning strategy, we focus on tuning the competitive parameters between read and write, such as the heap proportion assignment for write workload (e.g., *memstore.upperLimit*) and read workload (e.g., *block.cache.size*) according to the read/write proportion in the mixed workloads.

C. Policy-driven Configuration Management

1) Workload Aware Configuration Policies

We develop three categories of configuration policies in response to the three typical types of NoSQL workloads:

Configuration optimization for *write-intensive workloads:* Table IV shows an example set of parameters which are critical for performance tuning of write-intensive workloads. We provide the recommended settings by PCM under the PCM-BL/WO/WM column for three subcategories of write-intensive workloads: PCM-BL (bulk loading), PCM-WO (write only) and PCM-WM (write mostly). PCM-BL and PCM-WO use very similar parameter settings in HBase due to the fact that HBase implements Insert and Update operations with same API (Put). For WM, as a small proportion read workload is added, we increase the heap size for read from 0.1 to 0.2 and decrease the same amount of heap for write to maintain the total heap for memstore and cache to be under 80% of the max heap size to avoid out of memory error.

Configuration optimization for *read-intensive workloads*: For read only (RO) and read mostly (RM) workloads, we provide PCM-RO and PCM-RM respectively. Given that point reads (Read) typically need just one block transfer while range reads (Scan) may need more disk I/O transfers depending on the range of the records to be scanned and the block size. We configure a smaller block size for PCM-Read and a relatively bigger block size for PCM-Scan. For read mostly workload, added a small portion of write workloads, we slightly change the heap utilization ratio between block cache and memstore. Table V shows the PCM recommended settings of those parameters which are sensitive to read workloads.

Configuration optimization for *read/write-mixed workloads:* For three subcategories of read/write mixed workloads, we define three configuration policies: PCM-MixRW, PCM-MixR, and PCM-MixW respectively, by focusing on the trade-offs of heap contention for write part and read part. Table VI lists the PCM recommended values for the subset of parameters which are identified as sensitive to read/write-mixed workloads.

TABLE IV. WRITE-SENSITIVE POLICIES

Parameters	Default	PCM-BL/WO/WM
heapsize	1GB	(0.5~0.75)×RAM = *X* GB
memstore.upperLimit	0.4	0.6/0.6/0.5
memstore.lowerLimit	0.38	0.58/0.58/0.48
block.cache.size	0.4	0.1/0.1/0.2
memstore.flush.size	128MB	128MB×*X*
memstore.block.multiplier	2	max(2, *X*)
compactionThreshold	3	3×*X*
blockingStoreFiles	10	(5~10) × *X*
region.split.policy	IncreaseTo UpperBound	PreSplit[a]/PreSplit+ IncreaseToUpperBound
max.filesize	10GB	max(10GB,dataset/(#reg)) / 10GB/10GB

[a] Pre-split the target table into *#RSs* × *N* regions and *N* relies on storage I/O speed

TABLE V. READ-INTENSIVE POLICIES

Parameters	Default	PCM-RO/RM
blocksize	64KB	Read (point):16~32KB Scan (range): 128~256KB
heapsize	1GB	(0.5~0.75)×RAM = *X* GB
block.cache.size	0.4	0.6/0.6/0.5
memstore.upperLimit	0.4	0.1/0.1/0.2
memstore.lowerLimit	0.38	0.08/0.08/0.18

TABLE VI. READ/WRITE-MIXED POLICIES

Parameters	Default	PCM-MixRW/MixR/MixW
heapsize	1GB	(0.5~0.75)×RAM = *X* GB
memstore.upperLimit	0.4	0.4/0.3/0.5
memstore.lowerLimit	0.38	0.38/0.28/0.48
block.cache.size	0.4	0.4/0.5/0.3
memstore.flush.size	128MB	128MB×*X*
memstore.block.multiplier	2	max(2, *X*)
compactionThreshold	3	3×*X*
blockingStoreFiles	10	(5~10)×*X*
region.split.policy	IncreaseTo UpperBound	PreSplit/PreSplit+ IncreaseToUpperBound
max.filesize	10GB	max(10GB, dataset/(#reg)) / 10GB/10GB

2) PCM System Architecture

PCM is designed to automatically manage the set of configuration policies such as those outlined above. Each configuration policy is defined with a set of adaptation conditions, such as workload characterization, dataset size, running cluster environment. Fig.2 shows the PCM system architecture consisting of five main components: Workload Monitor, Policy Adaptation Manager, Policy Executer, NoSQL Interface and Optimal Configurations. The *Workload Monitor* gathers workload state (e.g., read/write request counts) statistics as well as the cluster state statistics (e.g., #RSs) from the master of the running cluster (e.g, HMaster). Two types of workload statistics are collected: workload requests statistics (such as requests per second, read/write request counts) and workload runtime environment (such as used heap (max heap), number of living workers (e.g., RSs), number of online regions, number of storefiles, compaction progress, and et al. The *Policy Adaptation Manager* determines the workload type and which policy with tuned parameters to be used according to the workload.

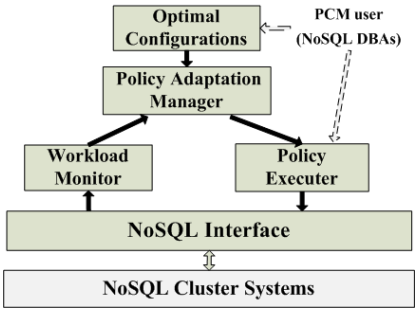


Fig. 2. Policy-driven Configuration Management: System Architecture.

The *Policy Executer (Executer*) setups and refines the configuration for the running cluster according to certain policy from the Policy Adaptation Manager. The *NoSQL Interface* enables Workload Monitor and Policy Executer to directly interact with NoSQL systems. The *Optimal Configurations* show the effective tuned configurations under various workloads by tuning the parameters to find the best setting for each independent NoSQL system. Another property of PCM is that we design and implement PCM as an open system to allow the administrators of NoSQL systems to insert new configuration policy and to update and replace existing configuration policy.

The functional components of PCM cooperate to accomplish the following five tasks: (1) Cluster state collection: the Monitor gathers the cluster state form the running cluster; If the target database is empty, then the Manager will setup the database with policy PCM-BL to prepare bulk load the target database. (2) Workload state collection: after the target database is loaded, the Monitor starts to collect the workload state statistics and periodically delivers the collected data statistics to the Manager for further decision making. (3) Workload characterization: when the Manager has received the workload state statistics, it will characterize the workload based on the workload state statistics. For example, the read/write request ratios can be used to categorize the current workload into one of the three workload types. (4) Configuration policy adaptation: based on workload state statistics collected periodically by the Monitor, the policy adaptation manager identifies the workload type and create new policy or refine existing policy. (5) Configuration Refinement: when Executer detects new policy updates arrives, it will execute the new or updated configuration with the recommended parameter values.

IV. Experimental Evaluation

We evaluate the effectiveness of PCM from three perspectives: (1) Tune configuration parameters under different workloads to show the performance optimization that PCM policies can provide. (2) Evaluate the typical policies with different target dataset sizes, request distributions and organizations of databases to show PCM's validity for workload variety. (3) Use a bigger cluster with four times size of the small cluster to evaluate PCM's scalability.

We use HBase version 0.96.2 and Hadoop version 2.2.0 (including HDFS) in all the experiments. HBase and HDFS are running in the same cluster with HMaster & NameNode on master node, RegionServer & DataNode on each worker node. Two clusters are used in our evaluation:

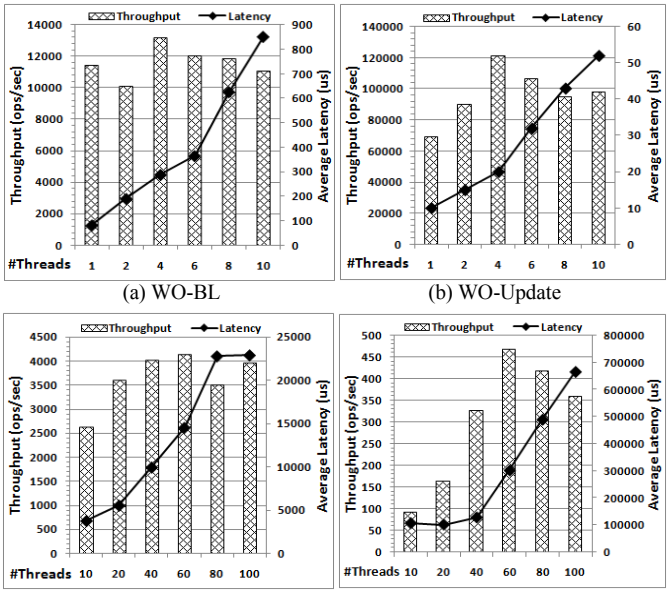


Fig. 3. PCM recommended concurrent #threads for typical workloads.

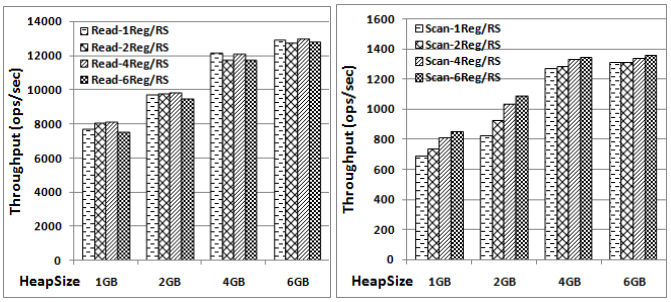
Cluster-small, consisting of 13 nodes: 1 node hosts both HMaster and NameNode as the master, 3 nodes host ZooKeeper cluster as coordinators and 9 nodes host RegionServers and DataNodes as the workers. **Cluster-large**, consisting of 40 nodes: 1 node as master, 3 nodes as coordinators and 36 nodes as the workers.

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 1 Gigabit Ethernet, run Ubuntu12.04-64bit with kernel version 3.2.0, and the Java Runtime Environment with version 1.7.0_45. We use YCSB version 0.1.4 to generate target types of synthetic workloads.

A. Tuning Configuration Parameters

This set of experiments uses cluster-small with a dataset of 10 million KV records of 1KB per record and uniform request distribution. We identify the number of concurrent threads that HBase client should use for achieving best overall throughput.

Fig.3(a) measures bulk loading throughput by varying the number of client threads. When the #threads for WO-BL is 4, the throughput is the highest and the average latency is good compared to other settings. Fig.3 (b) shows that when the #threads for WO-Update is set to 4, the throughput is the best with good average latency. Thus, we set 4 client threads as the PCM recommended #threads for write workloads (BL, WO, WM) on cluster-small in the rest of the experiments. Next, we examine the read intensive workloads (RO, RM). Fig.3 (c) and (d) show that when the #threads for both RO-Read and RO-Scan workloads set #threads to 60, the throughput is the best with relatively low average latency. Thus, PCM uses 60 threads as the recommended setting for read intensive workloads on cluster-small. In the next set of experiments, we conduct measurement study to show why PCM recommends different settings of some critical configuration parameters for different workloads as outlined in the previous section.



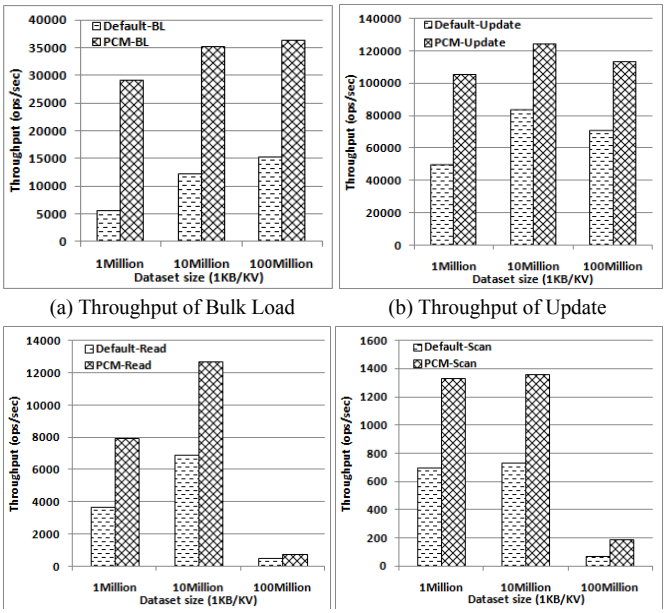
(a) Throughput of Read (b) Throughput of Scan

Fig. 4. Optimal heapsize and #regions per RS for read workloads.

We measure the throughput by varying the heap size setting for all three types of workloads. Due to the space constraint, we only include the results of read-intensive workloads. Also some detail on critical parameters and their impact on bulk loading performance of HBase are reported in [6]. Fig.4 shows the read throughput by varying the heap size from the 1GB default to 2GB, 4GB, 6GB. The target dataset is loaded to HBase using *PreSplit* bulk loading configuration. In addition, we also set the #regions to 1, 2, 4, 6 per RS for node level concurrency. We observe that bigger heapsize significantly improves the read throughput, but when heapsize exceeds 4GB, about half of the total RAM size (8GB) per RS, the throughput improvement become much smaller for both RO-Read and RO-Scan due to the workload pressure generated by one HBase client node is not enough. Thus, PCM recommends to use 1/2~3/4 of the total memory size for HBase heapsize to obtain high resource utilization, e.g., 6GB and 4GB for RO-Read and RO-Scan workloads respectively. In order to choose the best setting of #regions per RS for concurrency, we measured the throughput for different heapsizes under different #regions. Fig.4 (a) shows that the throughput for RO-Read workload is similar when we vary the #region per RS from 1 to 4, but when #regions per RS is increased to 6, the throughput starts to decrease for all heapsizes. However, for RO-Scan workload in Fig.4 (b), increasing #region per RS from 1 to 6 can improve the RO-Scan throughput consistently compared to RO-Read and the improvement is more pronounced for smaller heapsizes. Thus, by PCM recommendation, we use 4 region/RS for read-sensitive workloads here.

B. Workload Variety by Different Datasets

In this set of experiments, we vary target datasets from 1 million records to 10 million records and 100 million records. Fig.5 shows that the PCM offers consistently higher throughput compared to the HBase default for bulk load, update, read and scan workloads. Fig. 5 (a) shows that for write-intensive workloads, PCM achieves significantly better throughput than default with all the target datasets. Specifically, PCM-BL get 5.2x, 2.9x and 2.4x speedup in 1 million, 10 million and 100 million cases respectively compared with the HBase default (*IncreasingToUpperBound* region split policy). The reason is somewhat complex. One important objective for efficient bulk loading is to load the whole dataset into all the worker servers (RSs) evenly. An obvious optimization is to enable parallel processing and good load balance throughout bulk loading. However, the default policy implements the dynamic, threshold controlled incremental load balancing by *IncreasingToUpperBound* region split policy. Initially, only one initial region will handle



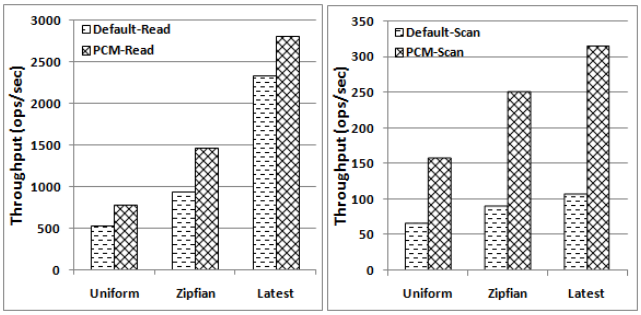
(a) Throughput of Bulk Load (b) Throughput of Update

(c) Throughput of Read (d) Throughput of Scan

Fig. 5. Evaluation results with different datasets.

bulk loading, and if all records can be loaded into a single RS without reaching *IncreasingToUpperBound* region split point, the default policy will load all data to only one region. Even when the *IncreasingToUpperBound* region split is triggered, if the balancer is not invoked, new coming records are still loaded to the current RS until new generated regions have been assigned to other RS by balancer. Thus, when the dataset is small or medium compared to the split point in the NoSQL cluster, a good portion of the cluster nodes are not used even the bulk loading has finished. This is why default policy lacks of parallelism and load balance during bulk loading.

Concretely, using the default policy, BL-1Million case only uses 1 RS and BL-10Million case only uses 4 RSs out of 9 RSs in the Cluster-small. Only when the dataset is much larger, say BL-100Million case, data is distributed to all 9 RSs with reasonable balance at the completion of the bulk loading. However, the throughput of bulk loading remains to be low for BL-100Million case due to imbalance at start stage and memory utilization inefficiency. In contrast, PCM recommends using Pre-Split policy to bulk load the target dataset across all RSs in the given cluster by distributing data to pre-split regions on all the RSs from the initial stage, and utilizing bigger heapsize with tuned memstore and hfile related parameters to achieve high memory utilization. Fig.5 (b) shows that the speedups are 2.1x/1.5x/1.6x for PCM-Update-1Million, 10Million, 100Million respectively. Then, Fig. 5 (c) and (d) show the throughput of RO-Read and RO-Scan respectively. And PCM-Read/Scan consistently outperforms Default-Read/ Scan. For 1Million and 10Million cases, PCM-Read/Scan is performed over the dataset bulk loaded based on the PCM-BL, thus the throughput improvements come from well-balanced data loads on each RS and bigger heap utilization by load much more records into memory to achieve accordingly higher cache hit ratio. So PCM-Read achieves 2.2x/1.8x speedup while PCM-Scan achieves 1.9x/1.9x speedup for 1Million/10Million cases respectively. Then, for 100Million case when target dataset becomes much larger, both default and PCM configuration cases get much lower throughput compared with



(a) Throughput of Read (b) Throughput of Scan

Fig. 6. Evaluation results with different request distributions.

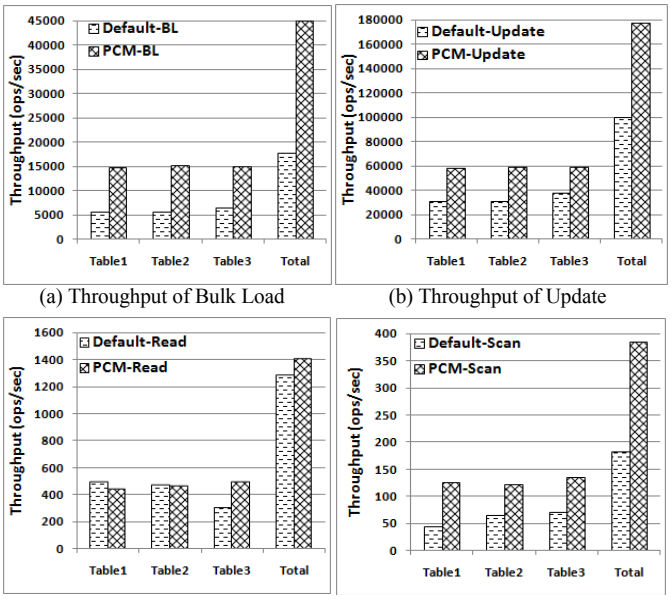
the small and medium cases during point and range read tests. This is primarily due to the fact that only very small part of the target dataset can be loaded into the cache, which sharply decreases the hfile block cache hit ratio. Also bigger heap size is not useful for point read case due to the uniform request distribution such that the same record is hardly to be read twice. For range reads (RO-Scan), the range length is uniformly chosen between [1, maxScanLength] (the default maxScanLength in YCSB is 100). So when the chosen range length is greater than the number of KV records in a block, more blocks will be loaded into memory after a RO-Scan operation and thus bigger heap utilization can improve the cache hit ratio, which leads to the PCM-Scan case achieve a 2.7x speedup, much higher than the point read case (only 1.5x).

C. Different Request Distributions

In this experiment, we change the request distribution from Uniform to skewed distributions by Zipfian (some records with high probability) or Latest (most recently inserted records have high probability to be chosen) [5]. We use a target dataset of 100 million records for Uniform and Zipfian tests, and insert 10 million additional records for Latest tests. Given that most NoSQL systems (incl. HBase) use append-style log writes (write-optimized), different request distributions have less impact on write-intensive workloads compared to read-intensive workloads. So in this set of experiments, we focus on RO-Read/Scan workloads. Fig.6 (a) shows that PCM-Read outperforms Default-Read for all three request distributions. As expected, the throughput of skewed distribution Zipfian and Latest exhibit much higher throughput than Uniform due to the increased cache hit ratio. This is because popular KV records can be maintained in heap and repeatedly accessed with high probability. Especially for Latest skewed distribution case, the very popular records are more concentrated in the most recently inserted records for large dataset of 110Million (10Million is inserted). With additional bigger heap, PCM-Read-Latest has higher speedup than other cases. For range reads in Fig.6 (b), only the start key is chosen with skewed distribution and maxScanLength is uniformly chosen. Thus, the dataset to be accessed for range reads is much larger than point reads to gain more spatial locality. And PCM-Scan achieves much higher speedup (2.7x/2.5x/2.9x) than PCM-Read (1.5x/1.6x/1.2x) for Uniform/Zipfian/Latest distribution respectively.

D. Multiple Databases with Varying Block Sizes

In this set of experiments, we evaluate the effectiveness of PCM for multiple database scenario where a NoSQL system hosts more than one database in its cluster.



(a) Throughput of Bulk Load (b) Throughput of Update

(c) Throughput of Read (d) Throughput of Scan

Fig. 7. Evaluation results on multiple databases.

We create three databases with different data block sizes from 32KB (Table1) to 64KB (Table2) and 128KB (Table3), and the target dataset for each table is 10 million records with a total of 30 million records. We run workloads on 3 separate YCSB client nodes concurrently, with each sends uniform requests to one target table. We compare each client's throughput and the total combined throughput of the three client nodes running with Default and PCM configurations. Fig.7 (a) and (b) are the results for write-sensitive workloads. PCM outperforms Default on each client node and the total throughput of PCM achieves 2.5x/1.8x speedup for BL and Update respectively. This indicates that PCM write workloads related configuration recommendation for single database also works well for multiple databases. For read-intensive workloads shown in Fig.7 (c) and (d), PCM outperforms Default at RO-Scan workloads under each client with different block sizes and the total aggregate throughput. For RO-Read in Fig.7 (c), PCM-Read outperforms Default-Read at database with large block size of 128KB (Table 3). However, for small block size of 32KB (Table 1) case, PCM-Read has slightly lower throughput than Default-Read. This is because point read request distribution is Uniform and cache hit ratio is very low, thus bigger heap has low utilization and does not help improving the throughput.

E. PCM Scalability

In this last set of experiments, we use the cluster-large with 36RSs to evaluate the scalability of PCM compared with the results from cluster-small with 9RSs. As the cluster-large is four times bigger than the cluster-small size, we use 4 YCSB client nodes concurrently to generate pressure enough workloads on both two clusters with a target dataset of 100 million records. Fig.8 shows PCM performs well consistently under various workloads for cluster-large and the results show PCM-36RS cases achieve 3.5x/4.0x/3.0x/3.6x speedup respectively for BL/Update/Read/Scan workloads, compared to cluster-small cases. This indicates that PCM maintains the horizontal scalability of HBase well and achieves almost linear

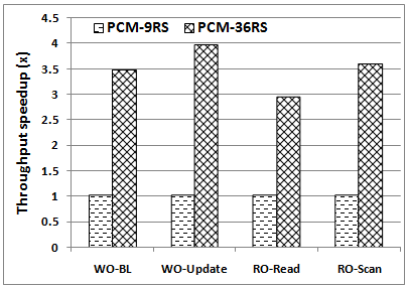


Fig. 8. Throughput speedup results for PCM scalability.

scalability ([3.5x, 4.0x, 3.0x, 3.6x] ≈ [0.87x, 1.0x, 0.75x, 0.9x] × 4) by just increasing the worker nodes (#RSs) for the running cluster.

V. Related Works

NoSQL evaluation: YCSB framework developed by Cooper et al. [5] is designed to generate representative synthetic workloads to compare the performance of NoSQL data stores for HBase [2], Cassandra [12], PNUTS [20], and a simple shared MySQL implementation. Patil et al [11] extends YCSB and builds YCSB++ to support advanced features for more complex evaluation of NoSQL systems, such as eventual consistency test. Both YCSB and YCSB++ use the default average configuration to evaluate the target NoSQL systems, instead of focusing on optimizing the configuration of underlying target systems. Our work on PCM focuses on policy-driven configuration optimization for NoSQL systems under representative workloads. This paper illustrates the design of PCM through enhancing HBase configuration to improve throughput performance of read and write workloads.

NoSQL (HBase) optimization: Cruz et al [7] present a framework to achieve automated workload-aware elasticity for NoSQL systems based on HBase and OpenStack. This work only considers very limited HBase parameters tuning such as heap and cache size. As we have shown in the bulk loading example, simply increasing heap and cache size without memstore parameters tuning will not help write-sensitive workloads and can even hurt performance. Das et al [10] implements G-Store based on HBase to provide efficient transactional multi-key access with low overhead. Nishimura et al [21] proposed MD-HBase to extend HBase to support advanced features such as multi-dimensional query processing. These functionality optimizations are orthogonal to our work on configuration optimization. Harter et al [3] present a detailed study of the Facebook Message stack to analyze HDFS and HBase, and suggest adding a small flash layer between RAM and disk to get performance improvement. This kind of improvement can also be helpful to PCM system.

VI. Conclusing Remarks

We have presented a policy-driven configuration management framework, called PCM, for disk-resident NoSQL systems such as HBase. PCM can analyze the range of configuration parameters those may impact on the runtime performance of NoSQL systems and make the parameter tuning recommendations for different workloads in form of configuration policies. We show that the configuration optimization recommended by PCM can enable the NoSQL system such as HBase to run much more efficiently than using the default settings for both the individual worker node and the entire cluster of different sizes in the Cloud. Although this paper uses HBase as the main example to illustrate the PCM development and tune the parameters related to the whole I/O stacks involved in memory, storage and network I/Os, the workload-adaptive and policy-driven configuration management principles also apply to other NoSQL systems. Even for main-memory-based NoSQL system such as Redis, the PCM framework can optimize the configuration management effectively.

Acknowledgment

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 NUDT are partially supported by the NSF of China under Grant Nos. 61433019 and U1435217. 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.

References

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

[2] Apache HBase, http://hbase.apache.org/.

[3] T. Harter, D. Borthakur, S. Dong, et al. “Analysis of HDFS Under HBase: A Facebook Messages Case Study”. USENIX FAST 2014.

[4] NoSQL wiki. http://en.wikipedia.org/wiki/NoSQL.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB,” ACM SoCC 2010.

[6] X. Bao, L. Liu, N. Xiao, et al. “HConfig: Resource Adaptive Fast Bulk Loading in HBase”. IEEE CollaborateCom 2014.

[7] F. Cruz, F. Maia, M. Matos, et al. “MeT: Workload aware elasticity for NoSQL”. Proceedings of ACM EuroSys 2013.

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

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. “The Hadoop Distributed File System”. IEEE MSST 2010.

[10] S. Das, D. Agrawal, A. Abbadi. “G-Store: A Scalable Data Store for Transactional Multi key Access in the Cloud”, ACM SoCC 2010.

[11] S. Patil, M. Polte, K. Ren, W. Tantisiriroj,et al, “YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores,” ACM SoCC 2011.

[12] A. Lakshman, P. Malik, and K. Ranganathan. “Cassandra: A structured storage system on a P2P network”. ACM SIGMOD 2008.

[13] 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.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, et al, “Bigtable: A Distributed Storage System for Structured Data,” USENIX OSDI 2006.

[15] G. DeCandia, D. Hastorun, M. Jampani, et al. “Dynamo: Amazon’s highly available key-value store”. ACM SOSP 2007.

[16] NoSQL databases. http://www.nosql-database.org/

[17] B. Atikoglu, Y. Xu, E. Frachtenbreg, et al. “Workload Analysis of Large-Scale Key-Value Store”. ACM SIGMETRICS 2012.

[18] L. George. HBase: The Definitive Guide. O’Reilly Media, 2011.

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

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, et al, “PNUTS: Yahoo!'s hosted data serving platform”, VLDB 2008.

[21] S. Nishimura, S. Das, D. Agrawal, et al, “MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services”, Proceedings of IEEE MDM 2011, Vol.1 pp 7-16

[22] K. Lee, L. Liu. “Efficient Data Partitioning Model for Heterogeneous Graphs in the Cloud”, IEEE SuperComputing 2013.

[23] H. Mao, H. Zhang, X. Bao, et al. “EaSync: A Transparent File Synchronization Service across Multiple Machines”, NPC 2012.