PAPER

A Configuration Management Study to Fast Massive Writing for Distributed NoSQL System

Xianqiang BAO[†a)], ***Student Member***, **Nong XIAO**[†b)], **Yutong LU**[††c)], ***and*** **Zhiguang CHEN**[††d)], ***Nonmembers***

SUMMARY NoSQL systems have become vital components to deliver big data services due to their high horizontal scalability. However, existing NoSQL systems rely on experienced administrators to configure and tune the wide range of configurable parameters for optimized performance. In this work, we present a configuration management framework for NoSQL systems, called *xConfig*. With *xConfig*, its users can first identify performance sensitive parameters and capture the tuned parameters for different workloads as configuration policies. Next, based on tuned policies, *xConfig* can be implemented as the corresponding configuration optimiaztion system for the specific NoSQL system. Also it can be used to analyze the range of configurable parameters that may impact the runtime performance of NoSQL systems. We implement a prototype called *HConfig* based on HBase, and the parameter tuning strategies for *HConfig* can generate tuned policies and enable HBase to run much more efficiently on both individual worker node and entire cluster. The massive writing oriented evaluation results show that HBase under write-intensive policies outperforms both the default configuration and some existing configurations while offering significantly higher throughput.

key words: *configuration management, optimization, massive writing, HBase, NoSQL*

1. Introduction

NoSQL systems have become vital components for many scale-out enterprises, due to their high horizontal scalability (also called *scale-out* scalability). NoSQL systems are typically key-value stores with nothing 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 [3] at Google; Dynamo [4] at Amazon; HBase [5] at Facebook and Yahoo!; Voldemort [6] at Linkedin and so forth. Among these NoSQL systems, the open-source HBase is a truly distributed, versioned, non-relational database modeled after Bigtable and widely used by many Internet enterprises. Instead of using Google File System (GFS) [7] as the distributed storage system, HBase is developed on top of the open source Hadoop Distributed File System (HDFS) [9], [10] and can inherently support MapReduce [11] to handle big data processing. For example, *Facebook Messages* [12] is a typical application at Facebook that handles millions of messages daily through HBase.

However, existing popular NoSQL systems, e.g. [5], [13], [14], rely on experienced system administrators to configure and tune the wide range of system parameters in order to achieve high performance under tuned configurations. While current big data management rely heavily on efficient access to their own NoSQL systems. For leading enterprises such as Google, Amazon and Facebook, they can afford researchers and experts to develop NoSQL systems, and tune the NoSQL systems for their own Internet services handing big data. However, it can be beyond buget for small companies to do so. Furthermore, some top enterprises such Google and Amazon do performance tuning efforts for their NoSQL systems as in-house projects, even the NoSQL systems such as Bigtable [3] and Dynamo [4] are not open source. Although there are many database tuning work such as [28]–[31], [40]–[42], but they focus on SQL based RDBMS systems and can not apply to NoSQL directly. As a result, a majority of NoSQL users without expert experience just use the default configuration for average performance. It is much more daunting challenge for developers and users with limited experience on NoSQL to be truly familiar with the 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 read/write-intensive applications? With the increased popularity of NoSQL systems, the problem of how to set up NoSQL clusters to provide good load balance, high execution concurrency and resource utilization becomes one significant challenge for NoSQL system administrators, developers and users [16].

In this paper, we present a general configuration management framework for NoSQL systems, called *xConfig*. First, we argue that it is essential to understand how

Manuscript received March 11, 2016.
Manuscript revised May 18, 2016.
Manuscript publicized June 20, 2016.

[†]The authors are with State Key Laboratory of High Performance Computing (HPCL), National University of Defense Technology (NUDT), Changsha, Hunan 410073, China.
[††]The authors are with School of Computer, National University of Defense Technology (NUDT), Changsha, Hunan 410073, China.
a) E-mail: baoxianqiang@nudt.edu.cn
b) E-mail: nongxiao@nudt.edu.cn
c) E-mail: ytlu@nudt.edu.cn
d) E-mail: chenzhiguang@nudt.edu.cn
DOI: 10.1587/transinf.2016EDP7104

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers

different settings of parameters may influence the runtime performance of NoSQL system under different workloads. We identify workload sensitive configurable parameters and capture the tuned parameters for a classification of workloads as configuration policies. Next, we analyze the impact of a range of configuration parameters and their interactions on the runtime performance of NoSQL systems in terms of write-intensive workloads. We show that simply changing some parameters from the default settings may not bring out the optimal performance. And the tuned parameter settings have complex dependencies among configurable parameters. Last but not the least, we discuss configuration tuning strategies for typical distributed NoSQL system such as HBase with specific performance related parameters. We conclude our work in three aspects as follows:

- Several inherent configuration problems are observed (Sect. 3) and we give out three main problem observations focus on typical workloads with detailed experimental results.
- A configuration management framework (*xConfig*) for NoSQL systems is designed (Sect. 4). And we present corresponding configuration tuning strategies for typical NoSQL system such as HBase which also apply to other distributed NoSQL systems (Sect. 5).
- A prototype called *HConfig* from *xConfig* framework is implemented based on HBase system. We accomplish a extensive evaluation of *HConfig* and the experimental results show the validity of our configuration management design of *xConfig* (Sect. 6).

The rest of the paper is organised as follows: Sect. 2 gives out an overview of NoSQL systems and then focus on the typical NoSQL system HBase; Related works are analyzed in Sect. 7 and Sect. 8 concludes the whole paper.

2. Overview

We give out the overview focusing on NoSQL concepts and a typical distributed NoSQL system–HBase first. Then we describe the main idea of our configuration management work.

2.1 NoSQL Concept and Typical System: HBase

Over the past decade, NoSQL systems have become the vital components for many scale-out enterprises due to the high horizontal scalability. NoSQL systems 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 through massive parallel processing. And these advantages of NoSQL can well handle the some challenges of SQL facing today. Specifically, behind the debate of NoSQL (Non-relational DBMSs) vs. SQL (RDBMSs) [17], [18] is the ACID vs. BASE argument [19], [20]. Although RDBMS with SQL can achieve complex transactions and queries, but in many scale-out Internet services these features are not the requirements and simple *Get/Put* operations are enough (e.g., social network applications, the scalability and flexibility in data structure are the most important requirements). Moreover, when turn to the Internet services consistency, the complete form of ACID consistency guaranteed by RDBMSs is not required and only weak consistency (e.g., eventual consistency) is needed [21], [22]. So NoSQL systems are always developed to achieve some specific goals for big data and real-time oriented Internet services.

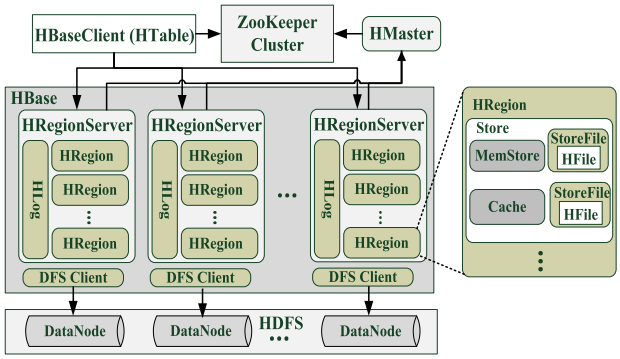


Fig. 1 HBase architecture overview.

During our research, we have experimented with many open source NoSQL systems, and in this work we mainly choose the typical distributed NoSQL system HBase for the discussion of its system design and data model. HBase [5] is an open source distributed key-value store developed on top of the Hadoop distributed file system HDFS [9], [10], and can inherently support MapReduce [11] to handle big data processing. For example, *Facebook Messages* [12] is a typical application at Facebook that handles millions of messages daily through HBase. As shown in Fig. 1, HBase consists of four major components: HMaster, ZooKeeper-cluster, 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 [23] 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 (*.META.* 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 (write buffer for HBase) and block cache (read cache for HBase). HBase supports two file types through the RSs: the write-ahead log and the actual data storage (*HFile*), and all the files are stored in HDFS.

2.2 Configuration Management Overview

Current existing popular NoSQL systems, such as [5], [13], [14], rely on experienced system administrators to configure and tune the wide range of system parameters. 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 to only achieve the average performance during the big data processing. Also it is much more daunting challenge for developers and users with limited experience on NoSQL systems to be truly familiar with the large set of parameters and understand how they should interact to bring out the optimal performance of a NoSQL system.

Our configuration management approach is to generate optimal configuration cases with tuned parameters to outperform the default configuration of NoSQL systems and then share the tuning experience. Base on our experience, usually different workloads need different parameters tuning to achieve adaptive optimal configurations. And the optimal configuration for a certain workload is called a configuration *policy* of the target NoSQL system. Then these policies from expert NoSQL administrators are captured as tuned parameters in individual configuration files and stored in the configuration management system. So when the target NoSQL system is running under a certain type of workload, the configuration management system can setup the corresponding policy to configure the NoSQL system with tuned parameter to achieve improved performance. Then the beginners of NoSQL systems can quickly obtain the performance improvements from these policies. Also, experts can generate configuration policies from their own target NoSQL systems and share the experience by the configuration management system easily.

3. Problems Statement: An Configuration View

Here we list our main problem observations focus on write-intensive workloads such as bulk loading with detailed experimental results and analysis as follows. We give out more details about the workloads in Sect. 5.2 and experimental setup in Sect. 6.1.

3.1 Unbalanced Workloads across RSs

The first observation from our experiments is the unbalanced workloads across the cluster of RegionServers when using the default configuration for bulk loading. Concretely, we bulk load 10 million records (1KB/record, 10fields/record, so 10GB total raw dataset) with default configuration. And we also complete a larger dataset case with 100 million records. Table 1 shows the file sizes distributed across all the RSs upon the completion of the bulk loading. In the scenario of loading 10 million 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 configuration of HBase 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 million records, we observe that all 9RSs are loaded with some portions of the input dataset but the data loading remains not well balanced across the cluster of 9 RSs (see Table 1).

Table 1 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

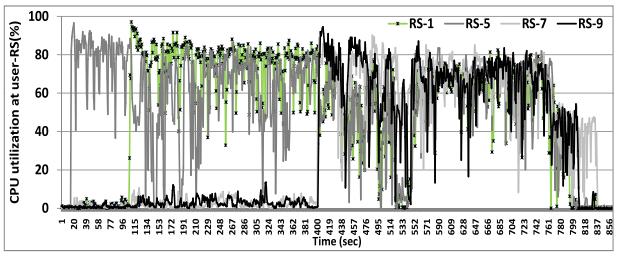
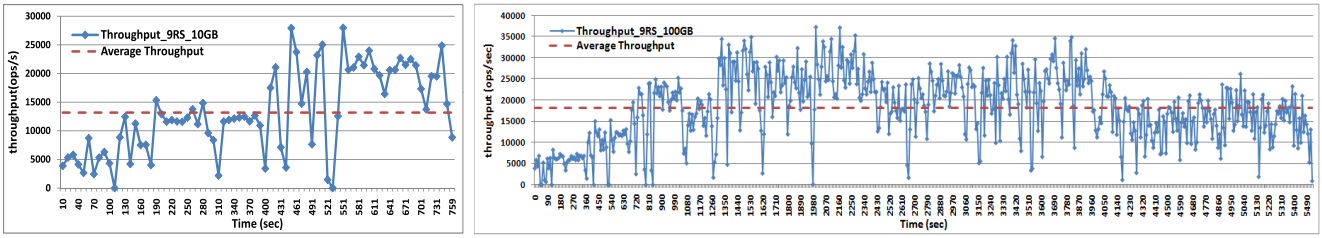


Fig. 2 CPU trace on 9RSs of bulk loading 10 million records.

To further study this result, from Fig. 2, we measure the CPU utilization for each of the four busy RSs which are loaded with input data for the 10 million records scenario. In addition, we also measure the real-time throughput (#operations/sec) for both scenarios in Fig. 3. Moreover, 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 in every stage. By examining the CPU utilization trace data collected by SYSSTAT [24] on the number of busy RSs, during each of the three throughput stages. From Fig. 2, 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, RS-7 & RS-9). When the bulk loading dataset is increased to the 100 million records, we observe from Fig. 3 (b) that the low throughput during the initial stage for the first 10 million records still exists, but the peak throughput for inserting later 90 million records is much more higher. This shows two facts: (1) When the dataset for bulk loading is big enough, the generated key-range based regions will be distributed across all the RSs after the initial stage and the records can be concurrently routed to all the RSs. (2) Moreover, the average throughput of bulk loading



(a) 10 million records (b) 100 million records

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

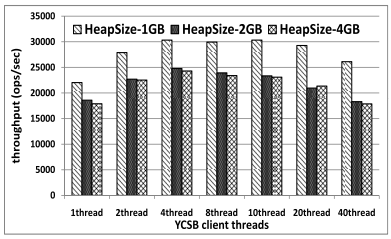
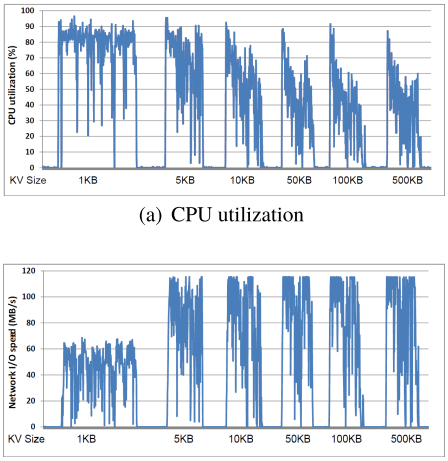


Fig. 4 Throughput with different heapsize.



(a) CPU utilization

(b) Network I/O speed

Fig. 5 Trace results on client node with varied key-value size.

larger dataset case is 37% higher, which is benefited from the concurrent data loading across all the RSs introduced by the incremental region splits and region reassignment.

3.2 Inefficient Resource Utilization on RSs

The second observation from our experimental results is the inefficient resource utilization across RSs and also on individual RS. First, from Fig. 3 (a), the bulk loading of 10 million records (1KB/record) is dealing with the raw dataset of 10GB (actual storage file size is a bit more than 14GB on HDFS), and the HBase cluster using a total RAM capacity of 72GB RAM from all nine RSs (8GB*9=72GB, see Sect. 6.1). However, there are only four out of nine RSs active and the average throughput of a single active RS is only about 5,000 ops/sec using 5MB/sec bandwidth, this is much less than the disk I/O bandwidth (50–100MB/sec) and network I/O bandwidth (peak value around 120MB/sec). When the bulk loading dataset is increased to 100 millions of records (about 140GB actual data, larger than the total RAM size), 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. We already know one main cause is the incremental region split policy in default configuration.

To further understanding the cause for short pauses which lead to throughput oscillation, we per-split the target table in order to distribute records across all the RSs evenly, and prepare balanced workloads. Then, we perform some in-depth experimental measurements by varying certain heapsize and disk I/O related parameters. For bulk loading, Fig. 4 shows the throughput with the different heapsize used by each RS ranging from 1GB, 2GB to 4GB, the different client side threads ranging from 1, 2, 4, 8, 10, 20 to 40. And 4 threads case is better than 1, 2, 20 and 40 threads cases. Furthermore, the frequent flushes from MemStore to store (HFile) and consequently frequent minor compactions can be triggered due to non-tuned parameters of default configuration on individual RSs, which leads to bigger heapsize is useless or even hurt performance. These analysis results motive us to study the set of configuration parameters which can be turned out according to cluster resource and node resource.

3.3 Inefficient Resource Utilization on Client and Network

Both of the above two observations are focused on server side, while client side based parameters tuning also has significant impact on resource utilization on client node and network. And the third observation is from client and network under default or improper configurations. During these bulk loading tests with varied key-value size, from default 1KB to 5KB, 10KB, 50KB, 100KB and 500KB. From Fig. 5 (a), We sequentially bulk load target dataset with 1KB to 500KB key-value sizes and use horizontal axis “1KB” to “500K” to mean each test case. The CPU utilization of client node that hosts application and generates key-value records is significantly influenced by key-value size. And when the key-value size becomes bigger than 5KB, the CPU utilization becomes very inefficient. Meantime, from Fig. 5 (b), the network I/O speed can also be significantly influenced by key-value size. And when the key-value size is smaller than

5KB, the network utilization becomes inefficient, such as the "1KB" case, the network utilization only achieves around half of the full utilization such as "10KB" case. Then, considering both Fig. 5 (a) and Fig. 5 (b), we need to find out well trade-off among resources such as CPU and network: when key-value size is smaller than 1KB, CPU has full utilization but network has only half of full utilization; when key-value size is bigger than 10KB, network achieves full utilization while CPU has only around 60% of full utilization. While the "5KB" case can achieve optimal trade-off to make both almost full utilization of CPU and network resources, also our evaluation results in Sect. 6 verify that well trade-off among client node and network resources achieves much better throughput speedup compared with default configuration. Moreover, besides key-value size and client running threads, other client side configuration related to write buffer size, doing batch or not and request distribution also have significant impact on the client and network resource utilization. And more details about the client side base tuning is discussed in Sect. 5.

4. Configuration Optimization Framework

In this section, a detailed configuration optimization framework (called *xConfig*) for NoSQL systems is given out with the following three aspects:

4.1 Design Objectives

xConfig is a general framework for NoSQL systems to manage their configurations setup and parameters tuning. Here the *x* of *xConfig* is used to denote a certain NoSQL system (e.g., for HBase is called *HConfig*). We focus on the following two design objectives: *(1) Uniform configuration management for NoSQL*, currently there are plenty of NoSQL systems [25], and many have their individual configuration files with lots of parameters. Here the uniform configuration management means that administrators with rich experiences can use *xConfig* framework to implement configuration management system for a target system, then other NoSQL users can use this system to manage configuration tuning for their running system. So *xConfig* is flexible to use the already implemented configuration management systems or integrate your own system into *xConfig* to share your tuning contribution. *(2) Adaptability to workload variety*, the adaptability is described as the recommended tuned configurations by *xConfig* can always match well with the running workloads and the tuned configurations can have better throughput performance compared with default configuration, no matter changing the workload runtime features from sever or client side.

4.2 System Architecture

Figure 6 shows the *xConfig* system architecture and the detailed design discussion about the six main components is based on HBase, also we believe the design experience

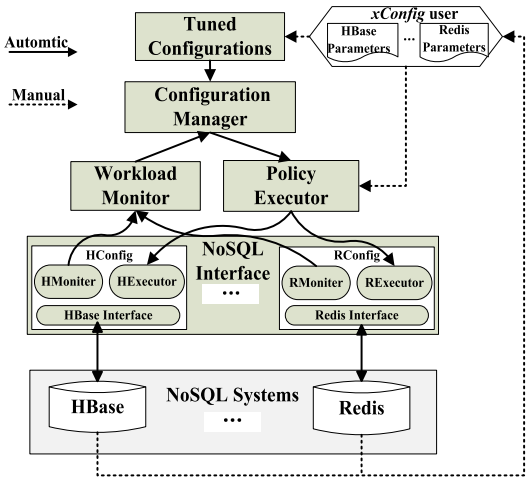


Fig. 6 Configuration management: system architecture.

can also apply to other distributed key-value based NoSQL systems.

(1) Workload Monitor (Monitor), periodically gathers workload state statistics from worker nodes or databases of a NoSQL system, and the cluster state statistics from the master node of the running cluster. As the statistics can be gathered by Monitor are depended on certain workload and system state trace components of specific NoSQL systems. For HBase system, there are always two types of workload statistics collected in the form of workload requests statistics, such as requests per second, read/write request counts; And the workload runtime environment, such as used heap (max heap), number of living worker nodes (here are RSs), number of online regions, number of store files and compaction progress (if LSM-Tree based).

(2) Configuration Manager (Manager), determines the workload type and which policy with tuned parameters to be used according to the workload type. Manger of *xConfig* needs to finish two main tasks during the decision-making process: (a) Determine the current workload type of the target NoSQL system. After Monitor periodically collects the workload state statistics and delivers to Manager, Manger picks out read/write requests statistics and calculates out the read/write ratio to determine the current workload type based on initial defined read/write ratio for typical workloads. (b) Determine the adaptive configuration policy for current workload. After working out the current workload type, and if the workload type switches to another different one, then Manger fetches the adaptive configuration policy from the prepared configuration polices.

(3) Policy Executor (Executor), setups and reconfigures the configuration for the target NoSQL system according to the applied policy. After Manager fetches the adaptive configuration policy and send it to Executor, Executor selects the target NoSQL system and distributes tuned configuration file or parameters to the related worker nodes. It is responsible for certain database schema changing or worker node reconfiguration. If the NoSQL system supports dynami-

cal database schema changing, Executor can re-setup the configurable parameters of certain database without stopping the data processing or rebooting the NoSQL system.

(4) NoSQL Interface (Interface), enables Monitor and Executor to directly interact with NoSQL systems. As there are lots of NoSQL systems and each system can have individual access interface, *xConfig* system needs to encapsulate the target NoSQL system interface and to be implemented as mainly two separated sub-components for Monitor and Executor. Interface for Monitor focuses on integrating the NoSQL system master interface to gather workload state statistics and target cluster running statistics. And interface for Executor focuses on invoking the database schema altering interface of the NoSQL systems with dynamical schema alteration support, and worker node daemon reboot as well as configuration file distribution interface.

(5) Tuned Configurations and xConfig User, Tuned Configurations are generated by target NoSQL system administrators with expert experience. And *xConfig* Users can be certain NoSQL system beginners who want to get benefits from existing tuned configurations or experts who want to share the tuning experience. In this work, Tuned Configurations are manually generated by *xConfig* Users. The main task of *xConfig* Users is to pick up the performance related configurable parameters and tune out these parameters for certain NoSQL system under different runtime.

4.3 Configuration Management Workflow

The functional components in *xConfig* cooperatively accomplish the following five steps as a tuning cycle: *(1) Cluster state collection*, the Monitor gathers the cluster state form the target running cluster; If the target database is empty, then the Manager setups the database with bulk loading policy to prepare bulk load the target empty 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 statistics, it will characterize the workload based on the workload state statistics. Specifically, the read/write request ratios can be used to categorise the current workload into one of the typical 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 Executor detects new policy updates arrives, it will execute the new or updated configuration with the recommended parameter values. Then a whole tuning cycle is completed with the above five steps and the following tuning cycles are completed with four steps from step *(2)*, until the database reloading happens and the tuning cycle restarts from step *(1)*.

5. HConfig Implementation with Tuning Strategies

In this section, we give out the configuration tuning strategies based on our experience. Each NoSQL system has its own implementation from *xConfig* framework and can be integrated into current *xConfigs* system and co-existed with other implementations.

5.1 Scope-Based Configuration Tuning Strategy

5.1.1 Performance Related Parameters

For key-value based distributed NoSQL systems, we pick out the performance related parameters from the following experience: (1) Pick out the parameters those have impact on memory, disk and network I/O performance of the whole I/O stack from applications using NoSQL system client library to NoSQL system server. Usually, memory related parameters focus on write buffer and read cache, and the total memory size can be used by NoSQL system such as heap size for JVM based NoSQL systems. Disk related parameters focus on how to flush the write buffer to disk, also the data structure used in NoSQL systems, such as LSM-Tree [26] based NoSQL system should consider the compaction processing related parameters which significantly impact the disk I/O performance. Network related parameters usually focus on records processing model such as batch or not, as well as key-value record size; (2) Clearly distinguish the scopes of configurable parameters and pick out performance related parameters from big to small scope. In this work, the scopes are first distinguished as server/client, then server scope can be further divided into four levels from big to small as cluster/worker/region/store (region is based on a sub key range of the whole key space, and region split related parameters has significant impact on performance for distributed NoSQL systems), also client scope can be further divided into two parallel levels as client library/application.

With these guidelines we pick out the performance related parameters from HBase system configuration file [5] as follows: The client requests from HTable are directly handled by the corresponding regions hosted on certain RSs of a HBase cluster. Thus, performance related server side based parameters are focused on RS scope parameters. And more details are listed in Table 2 (see *Server Side*) with mainly four scopes such as the whole cluster scope, a worker scope, a certain region scope as well as a specific store scope. 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: (1) *IncreasingToUpperBound* split policy, the default policy for HBase version 0.94 and later, which triggers region splits when region size meets the following threshold (called *Split Point*): $Split\ Point = min(N^3 * 2 * \alpha, \beta)$, N: the region number of a region server; α: the value of configured *memstore.flush.size*;

Table 2 HBase related configurable parameters.

Scope	*Server Side*	Description
Cluster	region.split.policy	To determine when a region to split.
Worker (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.multiplier	Block updates if memstore occupancy has reached *memstore.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	blockingStoreFiles	If more than #HFiles in any one Store then updates are blocked for the Region until a compaction is completed.
	compactionThreshold	When the #HFiles in any HStore exceeds this threshold, a minor compaction is triggered to merge all HFiles into one.
	compaction.kv.max	How many KVs to read and then write in a batch when do flush or compaction.
Scope	***Client Side***	**Description**
ClientLib (HTable)	AutoFlush	To do batch processing or not.
	WriteBufferSize	Both client side and server side use the same write-buffer size to transfer data.
Apps	KeyValueSize	The size of a record with N fields and each field is M bytes = N$*$M (Bytes).
	KeysDistribution	Key ranges generated in order by sort or hash function. Default is hash way with key ranges split uniformly by #RSs.

β: the value of configured *max.filesize* of a region. For example, in the default configuration, α is 128MB and β is 10GB, so the region split process can be carried out as follows:

{*Initial: new table allocates only one region by default;*
Split Point$_1$*:min(1*^{3}**2*128MB,10GB)=256MB;*
Split Point$_2$*:min(2*^{3}**2*128MB,10GB)=2,048MB;*
Split Point$_3$*:min(3*^{3}**2*128MB,10GB)=6,912MB;*
Split Point$_4$*:min(4*^{3}**2*128MB,10GB)=10GB;. . . . . ;*
The following Split Points all are 10GB.}

(2) *ConstantSize* split policy, which triggers region splits when the total data size of one store in the region exceeds the configured parameter *max.filesize*. (3) *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. (4) *Disabled* split policy (also called *Manual* split policy), which disables the auto split process so that region splits only happen by manual split operations. Besides, some important HBase client side based parameters can also have significant impact on system performance. We dig into some details about HBaseClient (HTable). And more details about client side parameters are in Table 2 (see *Client Side*).

5.1.2 Scope-Based Parameters Tuning

Distributed NoSQL systems are designed to run on a cluster of nodes. We argue that the configuration tuning strategies should be scope-based as cluster-aware and node-aware tuning. And we discuss the scope-based parameters tuning from the following three aspects with considering the implementation of *HConfig*:

(1) Cluster-aware tuning strategies: focus on tuning the configurable parameters which can improve the overall performance of the cluster. For *HConfig* system, we first identify a set of parameters those can be tuned to improve concurrency and load balance across the worker nodes (RSs). For example, the *PreSplit* strategy is designed to pre-split the target table to be loaded into independent and well balanced regions according to the number of RSs in the cluster, then distribute the data across the RSs based on the keys distribution. In addition, we need to further improve concurrent execution at each RS through multiple regions, we pre-split the large input dataset into P regions, $P = N \times \#RSs$. So that each RS will have N regions to concurrently handle read/write processes. 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. Moreover, 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 to handle these write-intensive workloads. And this enables high concurrency across RSs. While for read-intensive workloads, the concurrency is based on data distribution accomplished by write processes, so the well balanced regions distributed across RSs from *PreSplit* strategy still benefits read processes. Also read processes cause no region splits and other region reassignment activities so we just ignore the region assignment balance in cluster-aware tuning for read-intensive workloads.

(2) Node-aware tuning strategies: focus on tuning the parameters related to per-worker node resource utilization to improve the runtime performance of individual worker node. For the write-intensive workloads, we can delay the update blocking and the LSM-tree [26] 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 based on our experience), 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. In *HConfig* system, 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 which consume disk I/O much, and increase the threshold of *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 and high *memstore* heap contention. In contrast, for the read-intensive workloads, the tuning strategy focuses on cache hit ratio which has significant influence on read per-

formance. 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*&*lowerLimit*) and read workload (e.g., *block.cache.size*) according to the read/write proportion in the mixed workloads.

(3) Application-aware tuning strategies: In order to further improve performance based on application specific features, we can also take into account a set of client-side parameters, such as key-value size, write buffer size and workload running setup parameters. Our experimental results from *HConfig* show that these application specific features such as key-value size also play an important role in tuning out the optimal configurations. In this work, we focus on tuning the workload running pattern and the following parameters: *KeyValueSize*, the number of concurrent running threads on client side as well as the client nodes, and *WriteBufferSize*. 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.

5.2 Write-Intensive Configuration Policies

First, we define the *write-intensive* workloads in this work as following: *Bulk loading (BL)* workload loads the prepared dataset to an empty target database. BL requests are implemented as *Insert* operations (each *Insert* inserts a key-value record with a primary string *'Key'* and a number of string fields as *'Value'*); *Write-Only (WO)* workloads are further characterized into WO-Insert (*Insert* ratio = WO_Ratio, similar to BL) and WO-Update (*Update* ratio = WO_Ratio, each *Update* updates a key-value record by replacing one or several fields of the *'Value'*). *Write-Mostly (WM)* workloads refer to the workloads with a small amount of read workloads (less than WM_Ratio) added into the WO workloads.

Then we give out the tuned policies for write-intensive workloads: Table 3 (*Write-intensive Policies*) shows an example set of parameters which are critical for performance tuning of write-intensive workloads. We provide the recommended settings by *HConfig* 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* with same API (*Put*). For PCM-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.

Table 3 Tuned policies for HBase

Write-write-intensive 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	$128MB \times X$
memstore.block.multiplier	2	max(2, X)
compactionThreshold	3	$3 \times X$
blockingStoreFiles	10	$(5-10) \times X$
region.split.policy	IncreaseTo UpperBound	PreSplit / PreSplit+ IncreaseToUpperBound
max.filesize	10GB	max(10GB, dataset/(#reg))/ 10GB/10GB

5.3 *HConfig* System Implementation

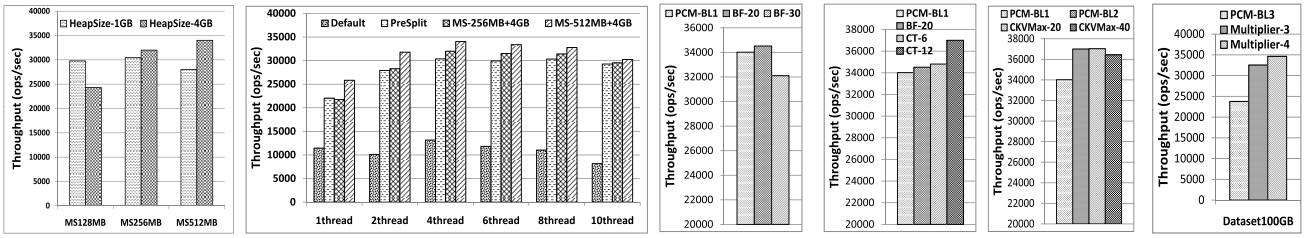
We develop the prototype *HConfig* system with about 1,500 lines mainly in Python from *xConfig* framework. The prototype is based on HBase system as it is widely supported from both the community and enterprises for handling big data processing. *HConfig* has three main parts during the implementation and we develop the prototype into a Python part, a Linux shell part and a HBase shell part. And the Python part is the main one to implement *xConfig* framework into *HConfig* system that comprises the core of the *Monitor, Manager, Executor, Tuned Configurations* modules. Meanwhile, each functional module has its own setup file to control the runtime parameters by *HConfig User*. Then we implement the *Interface* module that relates to the policy execution for certain NoSQL system HBase based on Linux shell and HBase shell, instead of using HBase HMaster API to achieve the lowest code changes.

6. Evaluation

We first give out the details about the experimental setup (Sect. 6.1). Then we focus on answering two main questions: (1) What is the speedup when we apply the parameters tuning under our strategies from server to client (application) side? (See Sect. 6.2). (2) Can the tuned configuration policies still work efficiently when workload scenario such as datasets and the number of databases (including database block size)? (See Sect. 6.3).

6.1 Experimental Setup

Each Node Setup: Each node of the cluster has single core (Dual socket) CPU operating at 2.6GHz with 4GB RAM per core (total 8GB RAM per node), and two SATA 7200rpm HDD. All nodes are connected with 1Gigabit Ethernet, run Ubuntu12.04-64bit OS with kernel version 3.2.0, and the Java Runtime Environment with version 1.7.0_45.



(a) HeapSize (b) MemStore (MS) (c) BF (d) CT (e) CKVMax (f) Multiplier

Fig. 7 Parameters tuning for write workloads (Bulk Loading).

HBase and HDFS Cluster Setup: We use HBase with version 0.96.2 and Hadoop with version 2.2.0 (including HDFS) in all the following experiments. And run HBase and HDFS in the same cluster to achieve data locality (HMaster and NameNode on manager node, RegionServer and DataNode on each worker node). We use a cluster during our evaluation works with 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.

YCSB Benchmark: In our experiments, workloads are generated by Yahoo! Cloud Serving Benchmark (YCSB) [27], and it is a framework for evaluating and comparing the performance of different NoSQL data stores. There are four baseline data manipulation operations of the workload generator YCSB: *Insert, Update, Read* and *Scan*, and here we focus on the write operations: each *Insert* operation inserts a key-value record with a primary string *'Key'* and a number of string fields as *'Value'*; each *Update* updates a record by replacing one field of the *'Value'*. During our tests, we generate target datasets with hash-based insert order, run all the workloads with unlimited target number of operations per second, and we set WO_Ration=100% and WM_Ratio=90%.

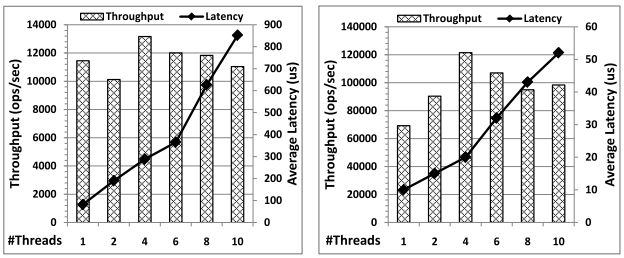
6.2 Configuration Parameters Tuning

We use a 10GB dataset with 10 million KV records (1KB per record) and uniform request distribution to identify the tuned parameters by default.

6.2.1 Server Side Parameters Tuning

We first focus on cluster-aware tuning strategy. Specifically, we pre-split the target table into 9 regions as there are 9 RSs in our cluster and set the region split policy to *ConsistantSizeRegionSplitPolicy* to generate *PreSplit* policy for further tuning out PCM-BL policy in *HConfig* system. From Fig. 7 (b), we can see the *PreSplit* significantly accelerates the throughput compared with *Default* configuration, the speedup is from *1.9x to 3.6x* with different client threads.

Then we focus on node-aware tuning strategy, and the following four tuning steps are based on *PreSplit*: (1) PCM-BL_1: *memstore.flush.size (memstore)*, we configure the



(a) WO-BL (b) WO-Update

Fig. 8 Recommended concurrent #threads for typical workloads.

heapsize from default 1GB to 4GB, then change the *memstore.flush.size* from default 128MB to 256MB, 512MB. Here we need to find out whether small *heapsize* with bigger *memstore.flush.size* can work well (in Sect. 3.2, we see performance reduction when using bigger heapsize with small memstore size) and the adaptive *memstore.flush.size* for bigger heapsize. From Fig. 7 (a), the small heapsize with bigger memstore.flush.size ({1GB, 512MB} case) leads to performance loss, and bigger heapsize with adapative *memstore.flush.size* ({4GB, 512MB} case) improves the performance and partly resolves the bigger heapsize hurting performance problem. Figure 7 (b) shows that the adaptive *memstore.flush.size* for bigger heapsize (4GB) is 512MB, and the improvement is 40-50% compared with just simply setting 4GB bigger heapsize cases, and also better than *PreSplit*. (2) PCM-BL_2: *blockingStoreFiles (BF)* & *compactionThreshold (CT)*, 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 LSM-Tree caused compactions. From Fig. 7 (c), 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 memstores too stressful to do flushes rather than to handle new arrived records. And from Fig. 7 (d), the optimal *compactionThreshold* is 12 as expect, while too bigger *compactionThreshold* can cause unacceptable compaction delay risk, we just use 12 as the optimal parameter. And speedup now becomes *2.81x* (select the best throughput case) compared with Default. (3) PCM-BL_3: *compaction.kv.max (CKVMax)*, from Fig. 7 (e), there is only a tiny speedup when increasing *compaction.kv.max* from the

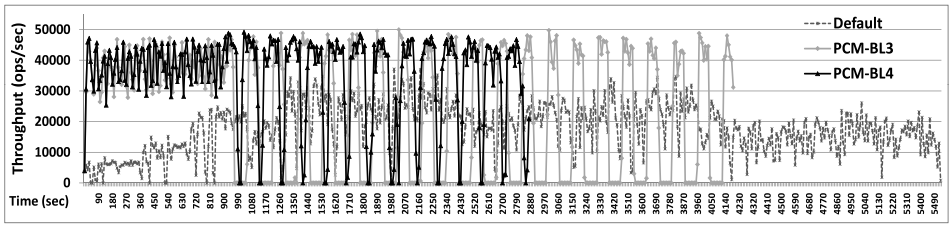


Fig. 9 Real-time throughput of bulk loading 100 million records.

default 10 to 20, the bigger setting with 40 even decreases the throughput, so we use 20 as the optimal setting. (4) PCM-BL_4: *memstore.block.multipiler (multiplier)*, disk I/O resources are already full used during the above three tuning steps. And when target loading dataset becomes much larger, although compaction is delayed, but it should not be delayed too much to cause unacceptable compaction delay risk. And during doing compactions and periodic memstore flushes 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 (see PCM-BL_3 in Fig. 9). So we increase the global memstore heap of one region to further delay updates blocking based on PCM-BL_3 by using bigger *memstore.block.multiplier* and change it from Default 2 to 3, 4. From Fig. 7 (f), the optimal *memstore.block.multiplier* is 4 and achieves 46% throughput improvement compared with PCM-BL_3, and we can see single periodic pause time is significantly decreased in Fig. 9 (see PCM-BL_4). Similar tuning steps to generate PCM-WO/WM focusing on these write-sensitive parameters.

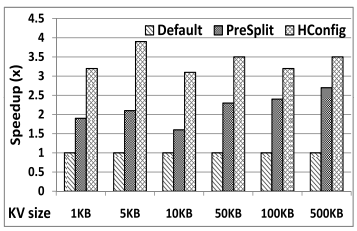
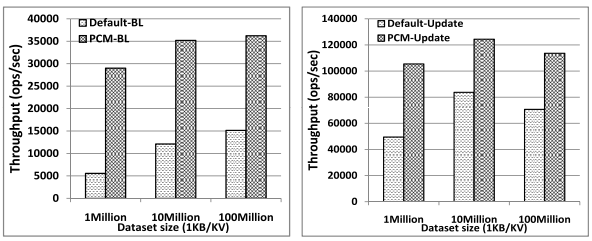


Fig. 10 Speedup for varied KV size.



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

Fig. 11 Evaluation results with different datasets.

6.2.2 Application Features Based Tuning

First, we focus on getting the optimal benchmark running threads. Figure 8 (a) measures the 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. Figure 8 (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 *HConfig* recommended #threads for write workloads (BL, WO, WM) on the cluster in the rest of the experiments.

Then we turn to other client parameters. And in the above tuning experiments, we use uniform requests distribution, so the main performance influence from benchmark setup is the key-value size (KV size). Here we change the KV size from 1KB to 5KB, 10KB, 50KB, 100KB, 500KB, and each record has 10 fields, also we use the default 12MB *WriteBufferSize* set by YCSB and batch way to handle requests. 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. 5 (Sect. 3.3), 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* can 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 for the cluster here. Moreover, from Fig. 10, *HConfig* system works well from 1KB to 500KB cases as maintaining with 3-4x speedup. 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 improve write performance by improving network I/O utilization with bigger *WriteBufferSize* based on PCM-BL_4 of the server side.

6.3 Tuning Adaptability

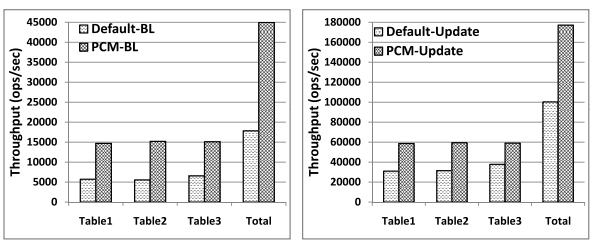
We verify the configuration tuning adaptability of *HConfig* with different datasets and multiple databases.

6.3.1 Different Datasets

We vary the target dataset from 1 million to 10 million and 100 million records. Figure 11 shows that the *HConfig* offers consistently higher throughput compared with default configuration. Figure 11 (a) shows that for write-intensive polices of *HConfig* achieve significantly better throughput than default with all the target datasets. Specifically, PCM-BL gets 5.2x, 2.9x and 2.4x speedup in 1 million, 10 million and 100 million cases respectively compared with the

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 nodes (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 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 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 configuration lacks of parallelism and load balance during bulk loading.

Concretely, using the default configuration, BL-1Million case only uses 1 RS and BL-10Million case only uses 4 RSs out of 9 RSs in this cluster. Only when the dataset is much larger, such as 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, *HConfig* recommends using *PreSplit* 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 setting bigger heapsize with tuned parameters to achieve high memory utilization. Figure 11 (b) shows that the speedups are 2.1x/1.5x/1.6x for PCM-Update-1Million, 10Million, 100Million respectively.



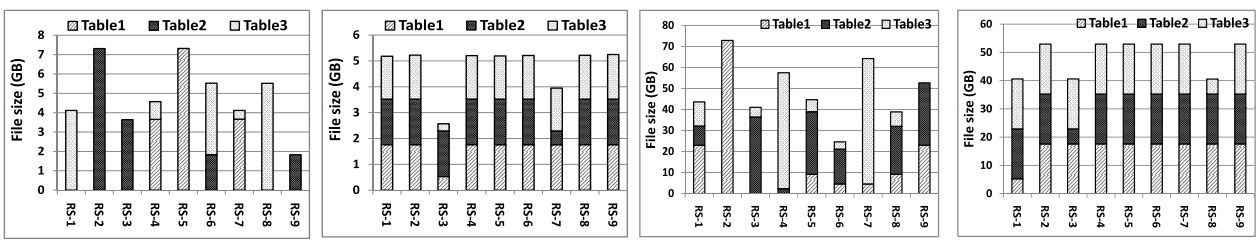
(a) Throughput of Bulk Load (b) Throughput of Update

Fig. 12 Evaluation results on multiple databases.

6.3.2 Multi-Databases with Variable Blocks

For multiple databases scenario, we create three databases with different data block sizes from 32KB (*Table 1*) to 64KB (*Table 2*) and 128KB (*Table 3*), and the target dataset for each table is 10 million records with a total of 30 million records. We run workloads on three separate YCSB client nodes concurrently, with each sends uniform requests to one target table. We compare the throughput of each client and the total combined throughput of the three client nodes running with Default and the policies of *HConfig*. From Fig. 12 (a) and Fig. 12 (b), *HConfig* outperforms Default on each client node and the total throughput achieves 2.5x/1.8x speedup for BL and Update respectively. This indicates that write-intensive policies of *HConfig* for single database also works well for multiple databases.

As multiple databases are more close to real production scenario in industry, then we dig into some more details behind the throughput results for the above multi-databases based experiments. As we discussed in Sect. 3.1 (see Table 1), there are only partial RSs are used during the single database bulk loading case that leads to unbalanced workloads across RSs. When turn to bulk load multiple databases currently, from Fig. 13 (a) (three tables and each hosts 10 million records), we can see default configuration also leads to very unbalanced store file distribution. Moreover, when we bulk load each table with 100 million records, from Fig. 13 (d), similar unbalanced workloads across RSs still exists, and the RS-2 almost hosts half of the whole dataset of Table 1 (100 million 1 KB KV records leads to a total $\approx$ 140 GB store file size and RS-2 hosts more than 70GB). While from Fig. 13 (b) and Fig. 13 (d), *HConfig* with *PreSplit* can leads to much better load balance for both small and large dataset cases, but the default hash function *FNVhash64* with *KeyPrefix* split policy still has inherent defects and can lead to lightweight unbalanced worker nodes such as RS-3 (*Table 1 & Table 3*) and RS-7 (*Table 2*) in Fig. 13 (b), also RS-1 (*Table 1*) and RS-3 (*Table 2*) and RS-8 (*Table 3*) in Fig. 13 (d). However, the unbalance in current *HConfig* is much more lightweight than Default. And one more interesting thing is that the key range generated by YCSB is uniform, when turn to highly skewed key range distribution, a more carefully *PreSplit* design is critical to achieve



(a) Default: 10 million records (b) HConfig: 10 million records (c) Default: 100 million records (d) HConfig: 100 million records

Fig. 13 Store file distribution details for multiple databases.

load balance. In *HConfig* system, we allow external data partitioning algorithms such as [32] to be plugged into the *PreSplit* policy.

7. Related Works

Ambari [1] is a project to achieve simplicity of Hadoop [9] management by developing a component for provisioning, managing and monitoring Hadoop cluster. Cloudera Manager [2] is a solution to get quickly automated deployment and configuration for Hadoop cluster. Also there are other tools such as [8] that developed in script to perform primary configuration tuning ([8] just focuses on memory resource tuning, and currently the users can use the Python script of [8] to calculate the memory related parameters in YARN, MapReduce 2 and Hive [9], [15]). These softwares and tools can significantly enables system administrators to manage Hadoop based cluster much simpler and quicker. Our work has the similar start point with [1], [2], [8], but we focus on the NoSQL layer systems and not the whole hadoop ecosystem. The prototype *HConfig* is similar to the subsystem for HBase of [1], [2], [8], and we focus on the holistic parameters tuning to form workload-aware optimal policies, while the script of [8] mainly focuses on the parameters tuning for memory. Moreover, the *xConfig* is a general design for the NoSQL systems with numerous configurable parameters. Administrators can implement *RConfig* based on *xConfig* framework for Redis [13] as future work.

Cruz et al. [37] 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 evaluation, simply increasing heap and cache size without memstore related parameters tuning (e.g. *memstore.upperLimit&lowerLimit, memstore.block.multiplier, et al.*) will not help write-intensive workloads and can even hurt massive write throughput. Das et al. [38] implements G-Store based on HBase to provide efficient transactional multi-key access with low overhead. Nishimura et al. [39] 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. [12] present a detailed study of the Facebook Message stack to analyze HDFS and HBase, and suggest to add a small flash layer between RAM and disk to get performance improvement. This kind of improvement can also be helpful to our system.

YCSB framework [27] is designed to generate representative synthetic workloads to compare the performance of NoSQL data stores for HBase [5], Cassandra [33], PNUTS [35], and a simple shared MySQL [18] implementation. The evaluation results in [27] are just use default configuration of mentioned NoSQL systems and only compare their average performance. Patil et al. [36] extends YCSB and builds YCSB++ to support advanced features for more complex evaluation of NoSQL systems, such as eventual consistency test. Also YCSB++ use the default configuration to evaluate the target NoSQL systems, instead of focusing on optimizing the configuration of underlying target systems. Our work focuses on configuration management for distributed NoSQL systems under write-intensive workloads.

Before NoSQL movement, there are many SQL based RDBMS tuning related work [28]–[31], [40]–[42]. Storm et al. [29] focus on self-tuning memory management in database system and propose workload-aware adaptive memory allocation in DB2 [43]. Also some products such as Oracle [30] and SQL Server [42] have implemented self-tuning memory management for high performance. Tran et al. [31] focus on database buffers self-tuning by using calculations with their proposed analytically-derived equation for buffer allocation. Wiese et al. [41] focus on the development of autonomic database tuning framework for RDBMS based on DB2 [43]. Cao et al. [28] propose a tool at the application code level to reengineer application code and table design to achieve additional tuning performance. Though our tuning work focuses on NoSQL systems, the above SQL based RDBMS tuning work has the guiding significance for our work. Such as the memory management including write/read buffer allocation related parameters tuning is still one of the most important tuning step for NoSQL systems. Also the application features based tuning can significantly impact the performance of NoSQL systems. The results from *HConfig* based on HBase give the verification for this viewpoint.

8. Conclusion

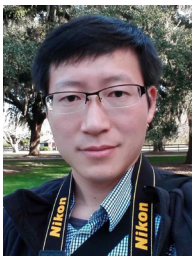
We have presented a configuration management framework named *xConfig* for disk-resident NoSQL systems such as HBase. With *xConfig*, its users analyse the range of configuration parameters related to the runtime performance, and then *xConfig* system makes the parameter tuning recommendations for different workloads in the form of tuned policies. Our HBase based prototype *HConfig* verifies that the tuning strategies work effectively, the tuned configurations for write-intensive workloads outperform the default configuration while offering significantly speedup. Although this paper uses HBase as the main example to illustrate the *xConfig* design and the tuning strategies related to the efficiency of memory, storage and network, but the configuration management framework also applies to other distributed NoSQL systems.

Acknowledgments

Authors from National University of Defense Technology (NUDT) are partially supported by the State High-Tech Development Plan (863 Program) of China under Grant No.2015AA015305, and the National Science Foundation (NSF) of China under Grant Nos.61433019, 61232003, 61402503 and 61303073.

References

[1] Ambari, "https://ambari.apache.org/"
[2] Cloudera Manager, "https://www.cloudera.com/products/cloudera-manager.html"
[3] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A distributed storage system for structured data," ACM Trans. Comput. Syst., vol.26, no.2, pp.4:1–4:26, June 2008.
[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," Proc. Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, New York, NY, pp.205–220, ACM, 2007.
[5] HBase, "http://hbase.apache.org"
[6] Voldemort, "http://www.project-voldemort.com/voldemort/"
[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," Proc. Nineteenth ACM Symposium on Operating Systems Principles, SOSP'03, New York, NY, pp.29–43, ACM, 2003.
[8] Hadoop Configuration Utils, "https://github.com/hortonworks/hdp-configuration-utils"
[9] Hadoop, "https://hadoop.apache.org/"
[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp.1–10, May 2010.
[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," Commun. ACM, vol.51, no.1, pp.107–113, Jan. 2008.
[12] A. Aiyer, M. Bautin, G.J. Chen, et al., "Storage infrastructure behind facebook messages: Using hbase at scale," IEEE Data Engineering Bulletin, vol.35, no.2, June 2012.
[13] Redis, "http://redis.io/"
[14] MongoDB, "http://www.mongodb.org/"
[15] Hive, "http://hive.apache.org/"
[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," 12th ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS'12, NY, USA, pp.53–64, 2012.
[17] M. Stonebraker, "Sql databases v. nosql databases," Commun. ACM, vol.53, no.4, pp.10–11, April 2010.
[18] MySQL, "http://www.mysql.com/products/enterprise/"
[19] D. Pritchett, "Base: An acid alternative," Queue, vol.6, no.3, pp.48–55, May 2008.
[20] S. Gilbert and N.A. Lynch, "Perspectives on the cap theorem," Computer, vol.45, no.2, pp.30–36, Feb. 2012.
[21] C. Xie, C. Su, M. Kapritsos, et al., "Salt: Combining acid and base in a distributed database," 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14), Broomfield, CO, pp.495–509, USENIX Association, Oct. 2014.
[22] P. Atzeni, F. Bugiotti, and L. Rossi, "Uniform access to nosql systems," Information Systems, vol.43, no.0, pp.117–133, 2014.
[23] ZooKeeper, "https://zookeeper.apache.org/"
[24] SYSSTAT, "http://sebastien.godard.pagesperso-orange.fr/"
[25] NoSQL Systems, "http://www.nosql-database.org"
[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," Acta Informatica, vol.33, no.4, pp.351–385, 1996.
[27] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," Proc. 1st ACM Symposium on Cloud Computing, SoCC'10, NY, USA, pp.143–154, ACM, 2010.
[28] W. Cao and D. Shasha, "AppSleuth: A Tool for Database Tuning at the Application Level," Proc. 16th International Conference on Extending Database Technology, EDBT'13, Genoa, Italy, pp.589–600, ACM, 2013.
[29] A.J. Storm, C. Garcia-Arellano, S.S. Lightstone, et al., "Adaptive Self-tuning Memory in DB2," Proc. 32nd International Conference on Very Large Data Bases, VLDB'06, Seoul, Korea, pp.1081–1092, VLDB Endowment, 2006.
[30] B. Dageville and M. Zait, "SQL memory management in Oracle 9i," Proc. 28th International Conference on Very Large Data Bases, VLDB'02, HK, China, pp.962–973, VLDB Endowment, 2002.
[31] D.N. Tran, P.C. Huynh, Y.C. Tay, and A.K.H. Tung, "A New Approach to Dynamic Self-tuning of Database Buffers," Trans. Storage, vol.4, no.1, pp.3:1–3:25, May 2008.
[32] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," Proc. VLDB Endow., vol.6, no.14, pp.1894–1905, Sept. 2013.
[33] Cassandra, "http://cassandra.apache.org/"
[34] A. Lakshman and P. Malik, "Cassandra: A structured storage system on a p2p network," Proc. Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, New York, NY, USA, p.47, ACM, 2009.
[35] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," Proc. VLDB Endow., vol.1, no.2, pp.1277–1288, Aug. 2008.
[36] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores," Proc. 2nd ACM Symposium on Cloud Computing, SOCC '11, New York, NY, USA, pp.9:1–9:14, ACM, 2011.
[37] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça, "Met: Workload aware elasticity for nosql," Proc. 8th ACM European Conference on Computer Systems, EuroSys '13, NY, USA, pp.183–196, 2013.
[38] S. Das, D. Agrawal, and A. El Abbadi, "G-store: A scalable data store for transactional multi key access in the cloud," Proc. 1st ACM Symposium on Cloud Computing, SoCC '10, NY, USA, pp.163–174, 2010.
[39] S. Nishimura, S. Das, D. Agrawal, and A.E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," A scalable multi-dimensional data infrastructure for location aware services," 12th IEEE International Conference on Mobile Data Management (MDM'11), pp.7–16, June 2011.
[40] B. Baryshnikov, C. Clinciu, C. Cunningham, et al., "Managing Query Compilation Memory Consumption to Improve DBMS Throughput," Proc. 3rd International Conference on Innovative Data Systems Research, CIDR'07.
[41] D. Wiese, G. Rabinovitch, M. Reichert, and S. Arenswald, "Autonomic Tuning Expert: A Framework for Best-practice Oriented Autonomic Database Tuning," Proc. 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, CASCON '08, Ontario, Canada, pp.3:27–3:41, ACM, 2008.
[42] Microsoft Corporation, "SQL Server 2008 R2 Books Online: Memory Management Architecture," "https://msdn.microsoft.com/en-us/library/cc280359(v=sql.105).aspx," May 2015.
[43] IBM DB2, "https://www.ibm.com/analytics/us/en/technology/db2/"



Xianqiang Bao received his B.S. degree from Huazhong University of Science and Technology (HUST) and M.S. degree from National University of Defense Technology (NUDT) both in Computer Science, China, in 2009 and 2012, respectively. During 2013–2015, he has been a joint Ph.D. student in College of Computing, Georgia Institute of Technology, USA. He is now a Ph.D. candidate in Computer Science, NUDT. His research interest includes data management and storage systems, network computing.

Nong Xiao received his B.S., M.S. and Ph.D. degrees in Computer Science from National University of Defense Technology (NUDT), China. He is now a professor in State Key Laboratory of High Performance Computing (HPCL), NUDT. He has been awarded the "Chang Jiang Scholars Program" professor by Ministry of Education of China, and the Distinct Young Scholar by the NSF of China. His current research interest includes large-scale storage system, network computing, and computer architecture.

Yutong Lu received his B.S., M.S. and Ph.D. degrees in Computer Science from National University of Defense Technology (NUDT), China. She is now a professor in School of Computer, NUDT. Her current research interest includes large-scale storage system, high performance computing, and computer architecture.

Zhiguang Chen received his B.S. degree in Computer Science from Harbin Institute of Technology (HIT), China, in 2007 and M.S. and Ph.D degree in Computer Science from National University of Defense Technology (NUDT), China, in 2009 and 2013, respectively. Now he is an assistant professor in the State Key Laboratory of High Performance Computing (HPCL), NUDT. His current research interest includes parallel file system and solid state storage system.