IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 2, 2017, accepted May 28, 2017, date of publication June 16, 2017, date of current version July 31, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2716441

ATH: Auto-Tuning HBase's Configuration via Ensemble Learning

WEN XIONG[1,2], ZHENGDONG BEI[1,2], CHENGZHONG XU[1,3], (Fellow, IEEE), AND ZHIBIN YU[1], (Member, IEEE)

[1]Chinese Academy of Sciences, Shenzhen Institutes of Advanced Technology, Shenzhen 518055, China
[2]University of Chinese Academy of Sciences, Shenzhen College of Advanced Technology, Shenzhen 518055, China
[3]Wayne State University, Detroit, MI 48202 USA

Corresponding author: Wen Xiong (wen.xiong@siat.ac.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2016YFB1000204, in part by the Major Scientific and Technological Project of Guangdong Province under Grant 2014B010115003, in part by the Shenzhen Technology Research Project under Grant JSGG20160510154636747, in part by the Shenzhen Peacock Project under Grant KQCX20140521115045448, in part by the Outstanding Technical Talent Program of CAS, and in part by NSFC under Grant 61672511.

ABSTRACT HBase is a distributed database management system and is becoming increasingly popular for applications that need fast random access to a large amount of data. However, it has a number of performance-critical configuration parameters, which may interact with each other in a complex way, making manually tuning them for optimal performance extremely difficult. In this paper, we propose a novel approach to auto-tune the configuration parameters for a given HBase application, called Auto-Tuning HBase (ATH). The key is an accurate performance model with low cost, which takes configuration parameters as inputs. To this end, we systematically explore different modeling techniques and decide to employ an ensemble learning algorithm to build the performance model. Subsequently, we leverage genetic algorithm to search the optimal configuration parameters for the application by using the performance model. As such, ATH can quickly as well as automatically identify a set of configuration parameter values to make the performance of the application optimal. We validate ATH in a cluster with ten nodes by using five typical applications from Yahoo! Cloud Serving Benchmark. The experimental results show that ATH can improve throughput by 41% on average and up to 97% compared with the default configurations. At the same time, the latency of HBase operations is reduced by 11.3% on average and up to 57%.

INDEX TERMS HBase, auto tuning, performance modeling, performance optimization, ensemble learning.

I. INTRODUCTION

HBase is a NoSQL column database system and widely used by internet-scale companies such as Facebook as well as manysmall-scale companies [14]. It can be used as a part of fundamental storage infrastructures. Fox example, the ''Facebook Messages'' [18] is built on the top of HBase, managing millions of messages every day. It can also be used as a storage system directly. For instance, in ''TimeTunnel'' [4] of Taobao, HBase is used to store real-time logging and feedback of advertisements. In fact, HBase supports a wide range of applications from business data analytics to scientific data computing.

However, HBase requires end users to determine up to 197 configuration parameters [33]. Although not all of them significantly affect performance, more than 20 do [6], [33]. A naive way towards configuration optimization is to manually set the value of each parameter to configure HBase, actually run an application with the configuration to measure its performance, and repeat this procedure for a number of times to identify the best configuration. This would work if the actual execution time is short as well as the number of needed executions is small. However, the number of needed executions for exploring the configuration space of HBase is huge because at least 20 configuration parameters need to be considered.

Moreover, we find these configuration parameters may interact with each other in a complex way, further complicating the configuration optimization issue. As a result, manually optimizing the configuration for a given HBase application without deep understanding of the internal mechanisms of HBase and that application is impractical. Therefore, an auto-tuning approach with low cost is

desirable. However, even if we automatically run a HBase application with each possible configuration parameter value combination to search the best configuration, the time would be extremely long because the number of the combinations is huge and each actual execution typically takes at least several minutes. A better alternative is to create a performance model which takes the configuration parameters as inputs and outputs a performance prediction. With this model, we can quickly search the optimal configuration automatically because the model can predict the performance of a HBase application with a certain configuration at the magnitude of milliseconds.

The key of such an approach is that the performance model for a given HBase application must be accurate enough as well as with low cost(e.g., the time used to build the model). On the one hand, if the model is not accurate enough, we can not find the optimal configuration. On the other hand, if the cost, the time for instance, to create an accurate model is too high, there might be no performance improvements. Given the complex interaction between HBase configuration parameters, it is difficult to build such models by using existing modeling techniques including analytical, statistic reasoning, and machine learning algorithms.

In this paper, we systematically explore the modeling techniques by trying typical statistic reasoning, machine learning, and ensemble learning algorithms. We find that with the same amount of training data, the performance model constructed by ensemble learning is much more accurate than those built by statistic reasoning and machine learning algorithms. Theoretically, ensemble learning combines the advantages of statistic reasoning and machine learning, and can thereby improve prediction accuracy [8]. We therefore finally choose ensemble learning to build performance models in this study. Note that we do not try analytical modeling technique because it always makes over-simplified assumptions, which is unsuitable for the performance modeling of HBase applications in practice.

After we have an accurate performance model, we employ genetic algorithm(GA) to automatically search the optimal configuration for a given HBase application. The GA takes the values of HBase configuration parameters and the corresponding performance predicted by the model as inputs, and outputs an optimal configuration for performance. We call this approach ATH (Auto-Tuning HBase's configuration) which can significantly reduce the configuration tuning effort for optimizing a HBase application.

In particular, we make the following contributions:

- We systematically explore modeling techniques and propose to leverage an ensemble learning algorithm, random forest [9], to build a performance model as a function of configuration parameters for a given HBase application.
- We propose to input configuration parameter values and the corresponding performance predicted by the model to the GA to automatically search the optimal configuration for a HBase application, and we call this approach ATH (Auto-Tuning HBase).
- We evaluate ATH by using five representative HBase applications from YCSB [14]. The results show that ATH improves throughput by 41% on average and up to 97% compared to default configurations. At the same time, the latency of HBase operations is reduced by 11.3% on average and up to 57%.

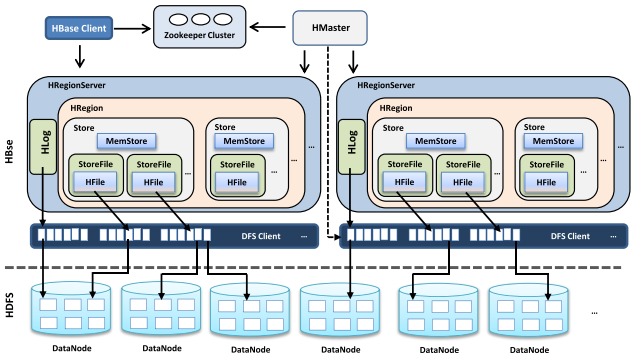

FIGURE 1. HBase architecture.

II. BACKGROUND AND MOTIVATION

In this section, we first describe the HBase architecture and data flow. Subsequently, we describe the motivation.

A. HBase ARCHITECTURE

Fig. 1 shows that HBase is built as a database layer atop the Hadoop Distributed File System(HDFS). HDFS handles machine failure, data replication, and placement, providing reliable and fault tolerant storage. HBase keeps data in indexed files on HDFS and handles real-time requests. As such, HBase is able to host very large data tables with billions of rows and millions of columns, which are distributed on thousands of machines. Consequently, it has been widely used in internet-scale companies such as Yahoo!, as well as many small-scale companies which do not have senior performance engineers.

In HBase, a big table is typically divided into a number of individual partitions called regions and each region is exactly hosted by a single server (called regionserver) serving clients with requested data. A regionserver runs a daemon called `HRegionServer` that makes a set of `HRegions` to clients. Each `HRegion` stores data for a certain region of a table, as shown in Figure 1. Inside a `HRegion`, there are a number of `Stores` and each holds a column family in that `HRegion`. Moreover, each `Store` has a `MemStore` which holds the in-memory modifications to the `Store`, and a corresponding `HFile` that provides the file format for HBase. In addition, each `Store` may have a number of `StoreFiles` used to store data. Note that all the changes to the `Stores` are stored in the object `HLog` and there is one `Hlog` per regionserver. Besides these core components, there are two auxiliary ones: `HMaster` and `ZooKeeper`, which are responsible for backup and error handling, respectively.

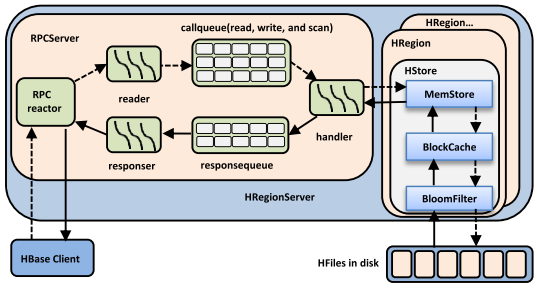

FIGURE 2. The data flow of HBase applications.

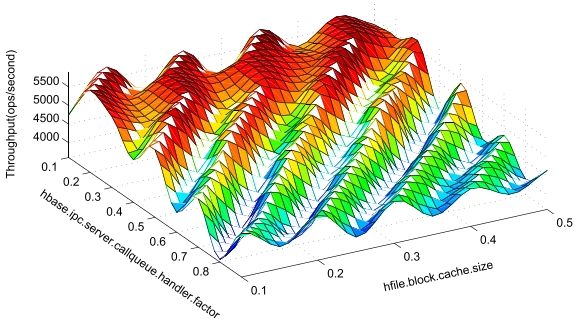

FIGURE 3. The influence of configuration parameters on the throughput of the HBase application *read-modify*.

Fig. 2 shows the core components involved in a HBase data flow. As can be seen, the communication between a client and a server is handled by a `RPC server` which consists of *RPC reactor*, *reader*, *callqueue*, *handler*, *responser*, and *responsequeue* components. The *RPC reactor* is in charge of handling requests from clients. The *reader* interprets the operations such as *scan* and *write* requested by a client. The *callqueue* is used to store the requested operations and there might be a number of *callqueue*s. The *handler* executes the requested operations and puts the results into *responsequeues*. The *responser* is in charge of sending results to clients. Note that when the *handler* executes an operation, two more components in the `HRegion` might be needed: *BlockCache* and *BloomFilter*. *BlockCache* is used to efficiently retain recently accessed data for subsequent reads. *BloomFilter* implements the bloom filter mechanism [10] which is used to test if a target row is contained in a given *HFile* or not.

HBase employs a number of configuration parameters to control the behavior of the components described above for flexibility. For example, the number of *callqueue*s is specified by parameter *hbase.ipc.server.callqueue.handler.factor*, and the size of *BlockCache* is determined by *hfile.block.cache.size*. Obviously, these two parameters are critical to the throughput and operation latency of HBase applications. In summary, HBase has up to 197 such configuration parameters controlling six aspects of HBase applications: `Client`, `CallQueue`, `MemStore`, `BlockCache`, `HStoreFile`, and `WAL`. Although not all of them significantly affect the performance of HBase applications, at least 20 parameters do [6], [33]. How to efficiently as well as optimally configure these parameters is therefore extremely important. However, it is very challenging to configure them to achieve optimal performance for a HBase application, as we will demonstrate in the next section.

B. MOTIVATION

To observe how configuration parameters influence the performance of HBase applications, we conduct an experiment by varying the values of two parameters — *hfile.block.cache.size* and *hbase.ipc.server.callqueue.handler.factor* — but keeping the values of other configuration parameters fixed to configure the *read-modify* application. The value of the former parameter increases from 0.1 to 0.9 with step size of 0.1 while that of the later varies from 0.1 to 0.5 with 0.05 as the step size. Note that *hfile.block.cache.size* is a percentage of the heap size of JVM which is used by HBase, see the details in Table 2. As such, we totally have 81 different configuration combinations. By running the *read-modify* (4 millions of read requests) application with each such configuration combination, we observe how the throughput varies.

Fig. 3 shows the experimental results. A couple of interesting observations can be made here. For one, the throughput varies dramatically when the values of these two parameters change. More concretely, the throughput varies from 3626.3 (ops/second) to 5914.2 (ops/second) across the 81 configuration combinations, which reflects a variation of 63%. Second, the relationship between the throughput and *hfile.block.cache.size* as well as that between the throughput and *hbase.ipc.server.callqueue.handler.factor* are non-monotonic. Instead, the throughput changes up and down when we increase the values of these two parameters. In such a case, one would painfully struggle on determining if he/she should increase or decrease the value of a parameter for better performance. Third, the two parameters interact with each other in a complex way with respect to throughput. As can be seen, HBase achieves the highest throughput, 5914.2 (ops/second), when *hbase.ipc.server.callqueue.handler.factor* is equal to 0.1 and *hfile.block.cache.size* is equal to 0.33. When we keep the former parameter fixed but change the later one to 0.4, the throughput decreases to 4415.2 (ops/second). Similarly, the throughput decreases to 4521.7 (ops/second) when we changes the former one to 0.12 while keep the later one fixed. When we further change the former one to 0.3, the throughput increases to 5500 (ops/second).

These results indicate the two parameters intricately intertwine with each other, which indicates that manually tuning the configuration parameters for a HBase application is extremely difficult, if at all possible. Furthermore, the later two findings present significant challenges for traditional modeling techniques such as statistic reasoning and machine learning techniques to build accurate performance models as functions of configuration parameters with low cost, which motivates this work.

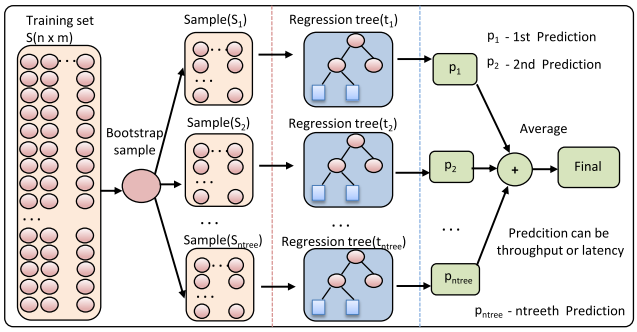

FIGURE 4. An overview of random forest.

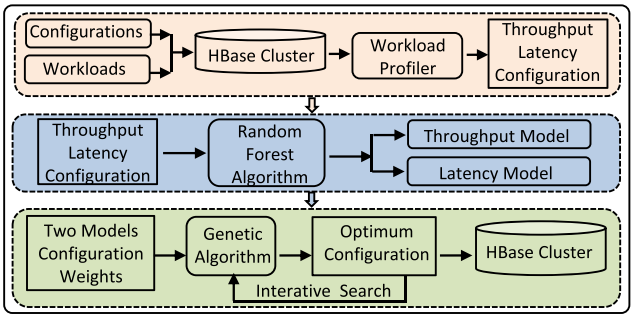

FIGURE 5. System architecture of ATH.

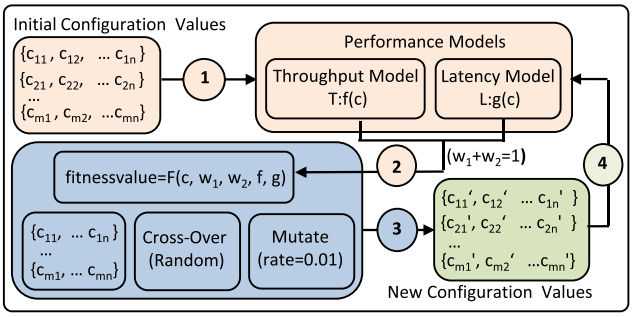

FIGURE 6. Searching work flow for identifying the optimal configurations of a HBase application.

C. RANDOM FOREST

Random forest [9] is an ensemble learning algorithm that can be used for prediction. It operates by constructing a multitude of regression trees at training time and then combines the outputs of individual trees to calculate the final output. A key feature of random forest is that it corrects for regression trees' tendency to over-fit to their training data. There are a couple of combination approaches such as majority voting for classification and average for regression.

Fig. 4 illustrates how to build a model by using the random forest algorithm. The training set S is a set of observations, which contains n observations in total. Each observation consists of a set of predictor variables (the HBase configuration parameters) and a corresponding response y (the throughput or latency of a HBase application) for them. The n observations are assumed to be independently and identically distributed (i.i.d.). A bootstrap sample is generated by uniformly sampling *ntree* instances from the training set S with replacement [16]. According to Fig. 4, the random forest algorithm operates as follows:

1) Draw *ntree* bootstrap samples from the training set S.
2) For each of the *ntree* bootstrap samples, grow an unpruned regression tree. At each node, randomly sample *mtry* of the predictor variables and choose the best split among those variables [25]. The result is stored in regression tree(t_i). There will be *ntree* trees at the end of this step.
3) Predict new data by aggregating the predictions of the *ntree* regression trees. In this case, the final prediction is average of p_i, ($i = 1, 2, \ldots ntree$).

Random forest learning is an extension of a bagging algorithm [8], which chooses the best split among all predictor variables at each node when growing an unpruned tree for each bootstrap sample. Although choosing predictors randomly is somewhat counterintuitive, it turns out to perform very well compared to many other classifiers, including discriminant analysis, Support Vector Machines (SVM) and neural networks [25]. Moreover, it is robust against overfitting [25] and it does not make any assumptions about the predictor variables. We therefore believe that random forest is a good candidate for constructing performance models for HBase applications.

III. ATH ARCHITECTURE

ATH is an automated performance tuning approach that adjusts the HBase configuration parameters for an application running on a given cluster to achieve optimized performance.

Fig. 5 shows the block diagram of ATH. When end users first run a HBase application, the ATH workload profiler collects the HBase configurations being used and the output performance metrics such as the throughput and latency being produced. In the scope of this study, we employ latency to represent 95 percentile latency because the latencies of different operations are different. Subsequently, the throughput or latency, and the corresponding configuration parameters are taken as input to the random forest algorithm to train performance prediction models. Finally, the throughput and latency model, weights of throughput and latency are taken as input to the GA. Fig. 6 illustrates the steps in GA for searching optimum configuration.

To build the models, we need to construct a training set S. S is a matrix, with each row being the following vector:

$$v_j = \{perf_j, c_{ij}, \ldots, c_{ij}, \ldots, c_{nj}\}, \quad j = 1, \ldots, m, \qquad (1)$$

with v_j the j^{th} observation, $perf_j$ the throughput or latency, and c_{ij} the i^{th} HBase configuration parameter of the j^{th} observation. n is the total number of HBase configuration parameters, and m is the total number of vectors in matrix S (observations or training examples). The training examples are collected in a dedicated cluster. In other words, there is no other workloads running concurrently on the same cluster, which avoids disturbing the performance measurements.

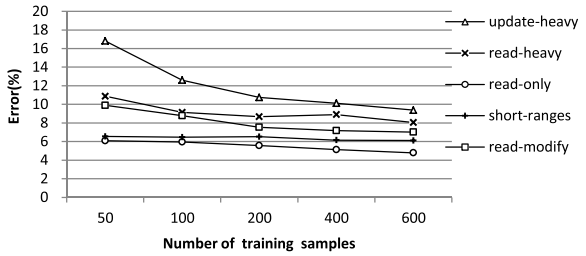

FIGURE 7. The accuracy variation of the throughput model built by ATH when increasing the size of the training set.

To collect vectors defined in Equation (1) for a HBase workload, we need to repeatedly run the workload a number of times, each time with a different configuration to form the training set (the matrix S). During the training phase, we need to make a trade-off between data collection time and model accuracy. A larger number of training configurations produce more accurate performance models but increase data collection and training time.

To understand this trade-off quantitatively, we have done the following experiment. We start to train the performance models using 50 HBase configurations, and we increase the training set by 50 vectors each time. All HBase configurations are randomly generated with each configuration parameter within its corresponding value range (ranges are shown in Table 2). Fig. 7 quantifies how accuracy is affected by the number of training examples, for the throughput of our five benchmarks. We find that all the models converge at the point where the number of training examples reaches 600. When the number of training examples increases from 200 to 600, the prediction error decreases from 7.80% to 7.06% on average. In practice, we can choose an appropriate number of training examples between 200 and 600.

After we have a model for throughput(or latency), we still do not know the optimum HBase configuration for a given workload with respect to throughput(or latency). Moreover, parameter tuning for HBase is a typical multiple-objective optimization problem because both the throughput and latency are important performance metrics. In this work, we employ the GA to automatically search the optimum configuration.

The GA takes the performance predictions produced by our models and the corresponding configuration parameter values as inputs to globally search the optimized configuration automatically, as illustrated in Fig. 6. In step ①, we take a set of randomly generated configuration parameter values as inputs to our throughput and latency model to predict the throughput and latency, respectively. The configurations and the corresponding throughput and latency then serve as inputs to the GA.

In step ②, we take the two models, two weights(one for the throughput, and the other for the latency), and the corresponding configuration as the inputs of the GA. As such, the `fitnessvalue` of the GA is produced by using the models. In the GA, the *crossover* operation randomly selects k configuration parameter values from one configuration set and $n-k$ values from another one, and then combines them into a new configuration set. In the new configuration, the probability for changing the value of a configuration parameter within its value range is controlled by the *mutation rate* which has a default value of 0.01.

In step ③, the GA outputs a new set of configuration parameter values which are passed to the performance models again. Steps ② and ③ may iterate a number of times until a configuration that yields the best overall performance (which is a sum of weighted throughput and latency) is found.

The overall HBase optimization framework is sufficiently fast to be used in practice. Collecting the profiling data (running a HBase application of interest with a small input data set for the 600 training configurations) takes less than two days. Moreover, as shown in Fig. 7, the prediction error is less than 10% when the number of training samples achieves 300 which take less than one and a half days. In this work, training performance models using the profiling data takes a few seconds while searching the optimum configuration using the GA takes time less than one minute.

Overall, we find that ATH takes less than two days to optimize the configuration for a given HBase application. This time overhead seems long but it is a one-time cost and is well justified as we target long-running HBase applications.

We now detail on the performance models for the throughput and the latency, and the GA to search the HBase configuration space.

A. PREDICTION MODEL FOR THROUGHPUT AND LATENCY

As mentioned before, we build two models. One is for throughput and the other is for latency:

$$throughput = f(c_1, c_2, \ldots, c_n) \quad (2)$$

$$latency = g(c_1, c_2, \ldots, c_n) \quad (3)$$

with $c_1, c_2, \ldots, c_n$ the values of the configuration parameters of HBase. Note that $f(\ldots)$ and $g(\ldots)$ are data models, which means there are no formula for them. $f(c_1, c_2, \ldots, c_n)$ and $g(c_1, c_2, \ldots, c_n)$ are obtained by using the random forest algorithm to train models based on the collected observations:

$$RF(throughput_k, c_{1k}, c_{2k}, \ldots, c_{nk}), \quad k \in [1, \ldots 600], \quad (4)$$

$$RF(latency_k, c_{1k}, c_{2k}, \ldots, c_{nk}), \quad k \in [1, \ldots 600], \quad (5)$$

with RF being the random forest algorithm as shown in Section II-C. $throughput_k$ is the throughput of the k^{th} observation, and $c_{1k}, c_{2k}, \ldots, c_{nk}$ are the values of the corresponding HBase configuration parameters for the k^{th} observation. Once we have constructed models for the throughput and latency, we can easily calculate the `fitnessvalue` for the GA.

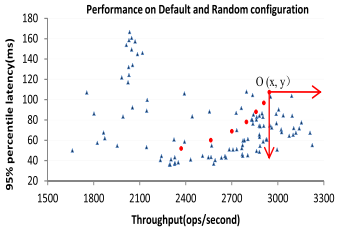

FIGURE 8. The throughput and latency of benchmark *read-only* with default and randomly selected configurations.

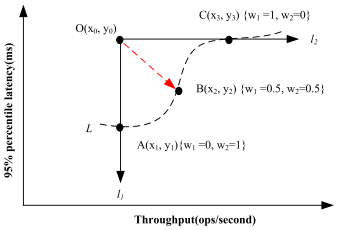

FIGURE 9. How to choose optimum points in throughput-latency plane.

B. SEARCHING THE HBase CONFIGURATION SPACE FOR OPTIMIZED PERFORMANCE

In this subsection, we first introduce the *throughput-latency* plane and the baseline. We then describe how to use this plane and baseline to search the optimum configuration for a given workload. Finally, we explain the reason why we select the GA as the search algorithm to identify the optimum configuration.

To observe how throughput and latency are influenced by configuration parameters, we run a HBase workload with different configurations and different number of client threads. For convenience, we call the execution of a workload with a certain configuration *workload-config* pair. Fig. 8 shows the results produced by a red group and a blue group of workload-config pairs for the YCSB workload *read-only*. The red-dots represent the throughput and latency of *read-only* with default configuration running with different number of client threads. Since the default configuration is always the same when *read-only* runs with different number of threads, each red-dot actually represents the throughput and latency of this workload with a certain number of threads. The blue-triangles represent the throughput and latency of *read-only* with randomly selected configurations running with variant number of client threads. As can be seen, there are many blue-triangles locate in the right-down area of each red-dot. This indicates that the default configuration of *read-only* does not necessarily achieve its optimized performance. In addition, the distribution of the blue-triangles implies that there is no configuration achieving the maximum throughput and the shortest latency at the same time.

Therefore, the optimum performance depends on the requirements of an end user. One may want maximum throughput but does not care about latency. Inversely, one may need to achieve the shortest latency but does not care about the throughput. In addition, one may need an optimized trade-off between the throughput and latency. To generally describe the different requirements of end users, we define the performance of a HBase workload as a weighted sum of throughput and latency as follows.

$$perf = w_1 \times throughput + w_2 \times latency \quad (w_1 + w_2 = 1) \tag{6}$$

with w_1 and w_2 the weights for throughput and latency, respectively. *throughput* is defined by Equation(2) and *latency* is defined by Equation(3). As such, our goal is converted to find the optimal *perf* based on end user's requirements.

To this end, we need a baseline. We take the point $O(x, y)$ shown in Fig. 8 as the baseline because it represents the maximum throughput when default configuration is employed. We then leverage Fig. 9 to describe how to find the optimum point in the *throughput-latency* plane where $O(x_0, y_0)$ is the baseline. The vertical line l_1 corresponds to the case that shortest latency is required($w_1 = 0$) and the horizontal line l_2 represents the case that maximum throughput is needed ($w_2 = 0$). For a given HBase workload running on a given cluster under a given w_1 and w_2, for example $w_1 = 0.5$ and $w_1 = 0.5$, the maximum *perf* is determined by the hardware limitation such as memory size and network bandwidth, the workload characteristics, and the configuration. Note that the hardware limitation and the workload characteristics can not be easily changed while the configuration does. In such a case, our goal is to find a configuration that makes the performance(*perf*) optimal. For example, we need to identify the configuration corresponding to the point $B(x_1, x_2)$ shown in Fig. 9 that represents the optimal *perf* for a HBase workload when w_1 and w_2 are both equal to 0.5. The $B(x_1, x_2)$-like points form a curve L.

We now describe how to identify the configuration for a point on curve L where the w_1 and w_2 are given. We first define a metric d as follows.

$$d = \sqrt{w_1(x_i - x_0)^2 + w_2(y_i - y_0)^2} \tag{7}$$

with x_i the throughput and y_i the latency of the i^{th} time when w_1 and w_2 are given. x_i and y_i can be calculated by Equation(2) and Equation(3), respectively. Subsequently, we employ a searching algorithm to identify the optimal configuration that makes the d maximum. Note that we normalize the values of the throughput and latency because they have different scales.

There exist many algorithms to search complex optimization spaces, such as recursive random search [36], pattern search [34], and genetic algorithms [23], [26]. Random recursive search is sensitive to getting stuck in local optima; pattern search typically suffers from slow local (asymptotic) convergence rates [34]. GA is a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover [26], and which is well-known for being robust against local optima [23].

TABLE 1. Workload and `target-set`s.

Workload	Operation(Mixture-Ratio)	Application Example	Target-Set(millon)
update-heavy(UH)	*read*(50%),*update*(50%)	user's session, which store the records of recent actions	10,20,40,80,120
read-heavy(RH)	*read*(90%),*update*(10%)	photo tagging, most operations are to read	10,20,40,80,120
read-only(RO)	*read*(100%)	application scenario is user profile cache	10,20,40,80,120
short-ranges(SR)	*scan*(95%),*insert*(5%)	application scenario is threaded conversation	0.1,0.2,0.4,0.8,1.2
read-modify(RMW)	*read*(50%), *readmodifywrite*(50%)	user database, user read and modify their activity	10,20,40,80,120

Our goal is to find the configuration for optimized performance of a HBase program from the global space of configuration parameters, which is a complex space to explore with many local optima. We therefore employ GA in this study. The values of HBase configuration parameters are passed to the GA as inputs and the d defined in Equation(7) is used as the `fitnessvalue`.

The GA is implemented by calling the function *rbga* in the R library *genalg* [27]. The *rbga* function has 11 parameters which are *stringMin*, *stringMax*, *suggestions*, *popSize*, *iters*, *mutationChance*, *elitism*, *monitorFunc*, *evalFunc*, *showSetting* and *verbose*. We set *popSize* (population size) to 200, *iters* (number of iterations) to 100, *mutationChance* to 0.01, *suggestions* to null, *elitism* to 40, *monitorFunc* to a monitoring function defined by ourselves showing the status of the GA iterations, and *showSetting* to false. The *evalFunc* is d defined in Equation(7). Note that *stringMin* and *stringMax* correspond to the minimum and maximum values of a configuration parameter, respecitvely. We will show the number of iterations needed by the GA to achieve optimized performance in Section 5.

IV. EXPERIMENT SETUP

A. HARDWARE PLATFORM

We use a HBase cluster consisting of 10 nodes: one serves as `HMaster` node, eight serve as `RegionServer` nodes, and one serves as `YCSB Client`. This cluster also serves as the underlying Hadoop Distributed File Sytem(HDFS) cluster. All the nodes are equipped with two 1 GB/s ethernet cards and are connected through two ethernet switches. One switch is used for global clock synchronization whereas the other switch is used for the communication in HBase cluster.

Each node has two 2 TB disks, 16 GB memory, and two Intel Xeon E5620 multi-core (Westmere) processors. The operating system running on each node is Ubuntu 12.04 with kernel version 3.2.0. The versions of HBase, Hadoop, and JDK are 1.0.3, 2.6.0, and 1.7.0, respectively. The monitoring system in HBase cluster is Ganglia 3.6.0.

B. BENCHMARKS

We employ Yahoo! Cloud Serving Benchmark(YCSB), a widely used benchmark frameworks, in this study. YCSB is designed to evaluate No-SQL systems such as HBase [3], PNUTS [13], Cassandra [1], Azure [11], CouchDB [2], SimpleDB [29], and Voldemort [5].

A key feature of the YCSB framework is that it is extensible and it supports easy definition of new workloads, as well as making it easy to benchmark new systems. In particular, YCSB employs five configurable attributions to define a workload:

1) `operation` — the basic operations in YCSB include *insert*, *update*, *read*, and *scan*. Note that each `scan` operation reads 100 successive records by default.
2) `record-size` — it defines the length of a record and decides the cost of each operation. The default value is 1024 bytes.
3) `mixture-ratio` — it defines the proportion of each operation in a HBase workload.
4) `request-distribution` — it is used to select the records to operate on. Distributions in YCSB include *uniform*, *zipfian*, and *latest*.
5) `target-set` — it defines the number of operations needed to be performed. It is specified by a configurable parameter: *operationcounts* in YCSB.

We select five typical YCSB workloads shown in Table 1 to evaluate ATH. Note that: (1) these five workload use *zipfian* as their `request-distribution`; (2) `record-size` is the default value:1KB; (3) we evaluate each workload with five different `target-set`s.

These workloads represent a sufficiently broad set of typical HBase application behaviors. For example, *read-heavy* contains 90% of *read* operations and it tests *BlockCache* in HBase. Its representative application is photo tagging which 90% of operations are to read tags and 10% of operations are to update tags. *update-heavy* contains 50% of *read* and 50% of *update* operations. It evaluates how *MemStore* interacts with *BlockCache*. Its typical application is to store the records of recent actions in a user session. *short-ranges* is a network-intensive workload and its typical application is threaded conversations used in forums of web sites.

C. CONFIGURATION PARAMETERS

According to the HBase reference guide [33], we consider 23 configuration parameters listed in Table 2. We divide these parameters into six categories: `Client`, `CallQueue`, `MemStore`, `BlockCache`, `HStoreFile`, and `WAL`. The fourth column provides default values of these parameters which are recommended by the reference guide and the last column shows the value range for each parameter in our experiments. Note that the value ranges of these parameters

TABLE 2. Description of the HBase configuration parameters.

Scope	Configuration Parameters	Description	Default	Range
Client	hbase.client.max.perregion.tasks	The max # of connections from client per single Region	1	1-5
	hbase.client.max.perserver.tasks	The max # of tasks from a HTable instance per regionserver	5	4-20
	ycsb.client.threads	The# of HBase clients	32	24-72
CallQueue	hbase.regionserver.handler.count	The# of RPC listener instances	30	20-80
	hbase.ipc.server.callqueue.handler.factor	Determine the# of sharing call queues	0.1	0.0-1.0
	hbase.ipc.server.callqueue.read.ratio	Determine the# of read queues	0.0	0.0-1.0
	hbase.ipc.server.callqueue.scan.ratio	Determine the# of scan queues	0.0	0.0-1.0
Memstore	regionserver.global.memstore.upperLimit	A threshold of memstore occupancy, block updates, force flushes	0.40	0.10-0.60
	regionserver.global.memstore.lowerLimit	A threshold of memstore occupancy,trigger flushes	0.35	0.10-0.60
	hbase.hregion.memstore.flush.size	A quota of heap size for memestore	64MB*2	16MB*(4-32)
Cache	hfile.block.cache.size	A quota of heap size for block cache	0.25	0.10-0.60
	hfile.index.block.max.size	The size of a index block	64KB*2	64KB*(1-10)
	hfile.block.index.cacheonwrite	A flag decides indexblock cache policy	false	false or ture
	io.storefile.bloom.block.size	The size of a single bloomblock	64KB*2	64KB*(1-10)
	hfile.block.bloom.cacheonwrite	A flag decides bloomblock cache policy	false	false or ture
	hbase.rs.cacheblocksonwrite	A flag decides data block cache policy	false	false or ture
HStoreFiles	hbase.hstore.blockingStoreFiles	A threshold about #HFiles, updates are blocked for the Region until a compaction finished	7	5-12
	hbase.hstore.compactionThreshold	A threshold about #HFiles, trigger a minor compaction	3	3-6
	hbase.storescanner.parallel.seek.enable	A flag of parallel-seeking	false	false or ture
	hbase.storescanner.parallel.seek.threads	The # of threads in parallel-seeking	5	4-20
WAL	hbase.regionserver.hlog.blocksize	block size for hlog in HDFS	16MB*8	16MB*(4-32)
	hbase.regionserver.maxlogs	A threshold about the #HLogs	32	16-48
JVM	java virtual machine heap size	The maximum amount of heap to use	512MB*8	512MB*(8-20)

might be cluster-specific because some ranges depend on the cluster hardware properties such as memory size and the number of cores in CPU.

V. RESULTS ANALYSIS

In this section, we first describe how to determine the *ntree* which is a model parameter of the random forest algorithm. We then evaluate the number of iterations needed by the GA to find the optimum configuration. Next, we evaluate the accuracy and error distribution of our performance models. Fourth, we report the speedup achieved by ATH. Fifth, we show and analyze the configuration parameter values for optimized performance for all the experimented workloads. Next, we discuss some insights we found, and finally, we report the overhead of ATH.

A. DETERMINING THE ntree

ntree(, see Section II-C) is the total number of trees used to construct our performance model which is an ensemble model for a YCSB workload. A larger *ntree* value leads to a higher accurate performance model but also results in longer model training time. We conduct experiments to determine a suitable for *ntree* for the throughput and latency model of each HBase workload. Figure 10 illustrates the experiments for the *update-heavy* and *read-heavy* as examples. As can be seen, the mean square error (MSE) converges once *ntree* exceeds 550 for *update-heavy* and 400 for *read-heavy*. Therefore, we set *ntree* to 550 and to 400 when we build throughput models for *update-heavy* and *read-heavy*, respectively. Similarly, we set the values of *ntree*s for *read-only*, *short-ranges*, and *read-modify* to 350, 450, and 500, respectively.

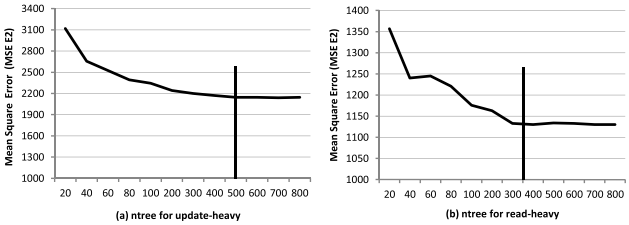

FIGURE 10. Determining the *ntree* parameter for *update-heavy* and *read-heavy*.

B. ITERATION NUMBER OF THE GA

As aforementioned, the GA iteratively searches the huge configuration space to find an optimum configuration for a HBase workload. The larger number of iterations are needed for convergence, the longer time it takes. Figure 11 shows how the GA converges for all the experimented benchmarks. As can be seen, a small number of iterations, say 30 to 45, are enough for all the benchmarks to achieve their optimized

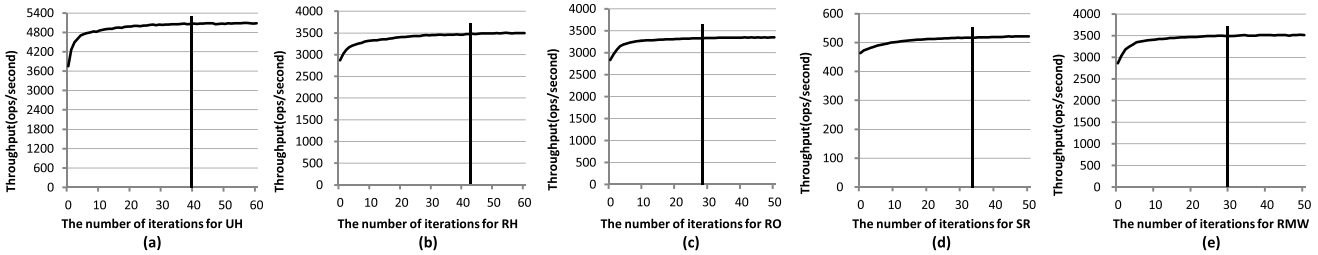

FIGURE 11. The number of iterations needed to find the optimum configurations for all experimented workloads.

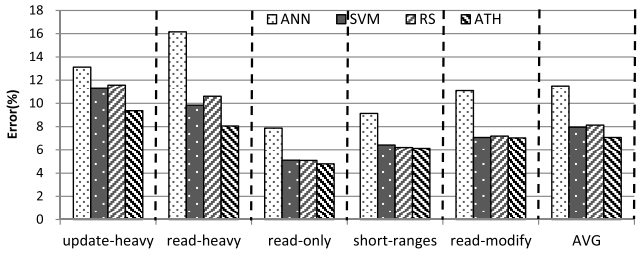

FIGURE 12. The model accuracy comparison between ANN, SVM, RS, and ATH. ANN–Artificial Neural NetWork, SVM–Supported Vector Machine, RS–Response Surface.

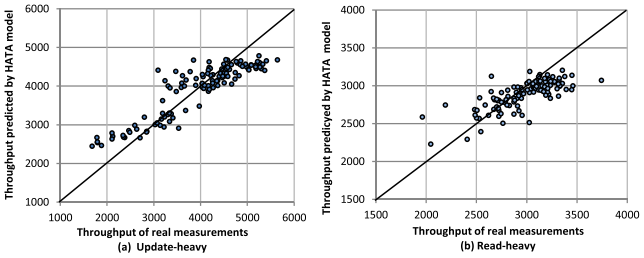

FIGURE 13. Error distribution for 120 randomly selected HBase configurations (the throughput predictions versus real measurements).

performance. Moreover, different benchmarks may need different number of iterations to achieve the optimized performance. For example, *read-only* needs 30 iterations to find the best optimizations while *update-heavy* needs 40 iterations, resulting in different optimization costs for different benchmarks.

C. MODEL ACCURACY

We now evaluate the accuracy of our performance models and compare them against those built by Supported Vector Machine(SVM) [24], Artificial Neural Network(ANN) [15], and Response Surface(RS) [17] because they have been used to optimize the performance of cluster computing systems. To this end, we randomly generate 120 configurations for each experimented benchmark. We run the benchmarks on the real HBase cluster with the generated configurations and measure their throughput and latency. We also use the performance models built by ATH, RS, ANN, and SVM to predict the throughput and latency. Subsequently, we leverage Equation (8) to calculate each prediction error and we therefore have 120 such errors for each performance model of each benchmark. Finally, we employ the average of the 120 errors of a performance model to represent the model accuracy.

$$err = \frac{|t_{pre} - t_{mea}|}{t_{mea}} \times 100\% \tag{8}$$

with err the prediction error, t_{pre} the predicted throughput or latency, and t_{mea} the measured throughput or latency.

Fig. 12 shows the accuracy of the models built by the four techniques. As can be seen, the performance models built by ATH for all the benchmarks are the most accurate ones. Even the highest error occurred for *update-heavy* is only 9.3%; the average error for all benchmarks is 7.1%. In contrast, the average errors of the models built by ANN, SVM,and RS are 11.8%, 8.1%, 8.3%, respectively. This indicates the performance models built by ATH are accurate enough to be used to search the optimum configurations. On the other hand, these results confirm that the models built by ensemble learning techniques are more accurate than traditional statistic reasoning and machine learning techniques, which also explains why we choose random forest to construct the performance models for HBase workloads.

D. ERROR DISTRIBUTION

As aforementioned, our model accuracy is the average error of the testing models. However, the average error might hide large errors for particular predictions due to outliers. To validate if there are many outliers, we now present the error distribution of our prediction models using scatter plots. In this subsection, we take throughput model as examples.

Figure 13 shows two scatter plots produced by 120 real measurements and 120 ATH predictions for benchmark *update-heavy* and *read-heavy* for 120 randomly selected HBase configurations. The X axis represents the real measurements and the Y axis denotes the ATH predictions of throughputs of the two benchmarks. This figure clearly shows that the models are fairly accurate across the entire HBase configuration space: all 120 data points for each benchmark are located around the corresponding bisector, indicating that the predictions are close to the real measurements. We observe a standard deviation of the predictions of 0.09 and 0.08 for the *update-heavy* and *read-heavy* models, respectively. In addition, we observe similar results for the other benchmarks. This indicates that there are not many outliers

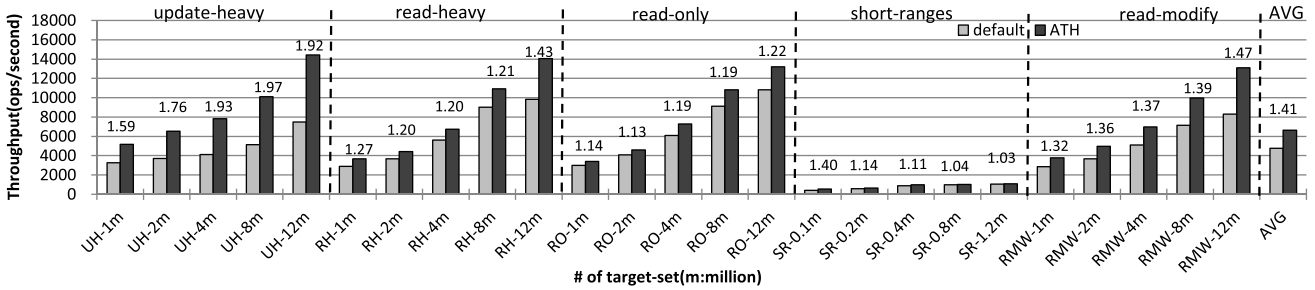

FIGURE 14. The speedup comparison between the default configuration and the ATH configuration when $w_1 = 1.0$ and $w_2 = 0.0$. for all the experimented benchmarks.

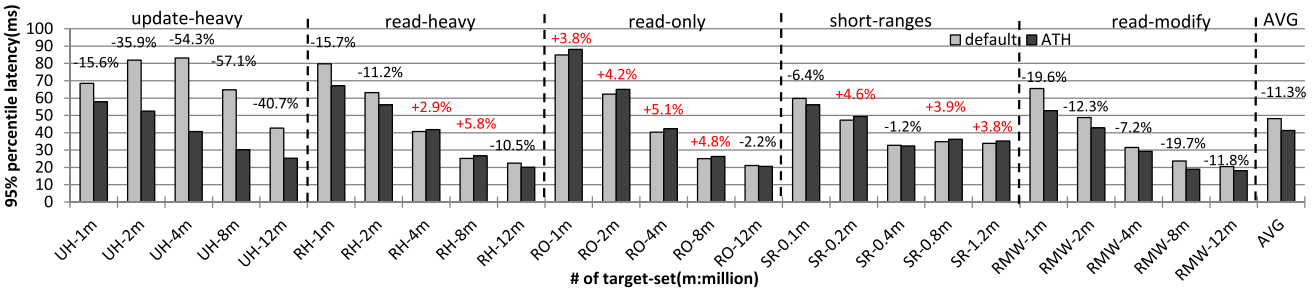

FIGURE 15. The 95th percentile latency comparison between default configuration and ATH configuration when $w_1 = 1.0$ and $w_2 = 0.0$. For *update-heavy*, *read-heavy*, *read-only*, and *read-modify*, we report the latency of the *read* operation while for *short-ranges*, we report the latency of the *scan* operation because this workload does not have read operation. Note that real latency of *scan* is ten times to the value in the Y axis.

in the predictions of our performance models, which is good for quickly finding the optimum configuration for a HBase workload.

E. SPEEDUP

We now evaluate the speedup achieved by ATH. Fig. 14 shows the speedups of the five experimented workloads, each with five different `target-sets`. The dark bars represent the throughput of workloads running with the configurations obtained by ATH while the gray bars show those of workloads running with the default configurations. There are several interesting observations to be made here.

For one, the throughput of all benchmarks running with the configurations obtained by ATH are higher than those of the same benchmarks running with the default configurations. Second, ATH significantly improves the performance of all the experimented benchmarks except *short-ranges*. The reason is that *short-ranges* is a network-intensive workload and network bandwidth becomes the bottleneck when the number of `target-set` is larger than 0.4 million, which makes the configuration tuning fail to improve the throughput further. The maximum speedup achieves 1.97× when *update-heavy* runs with the 8 millions of `target-set` and the average speedup for all the benchmarks is 1.41×. Third, the performance improvement for a benchmark made by ATH generally increases when its `target-set` increases. This is a very nice property for data analytics in big data era because one of the important features of big data is that the amount of data increases rapidly.

Fig. 15 shows the latency comparison between the default configurations and the ones obtained by ATH. As can be seen, ATH reduces the latency for most of the benchmarks from 1.2% to 57.1%. However, for a few benchmarks, the latency slightly increases (from 2.9% to 5.8%) when the configurations obtained by ATH are used. On average, ATH reduces the latency of all benchmarks by 11.3%.

We now discuss relationship between the throughput and latency improvements. Fig. 14 and Fig. 15 show that the improvements of throughput and latency are workload-specific. For example, for workloads *update-heavy* and *read-modify*, ATH improves throughput and reduces latency simultaneously. However, for workload *read-only* with a certain number of `target-sets` such as 1 to 8 millions, ATH improves throughput but increase latency. The same applies to *read-heavy* and *short-ranges*. These workload-specific properties leave much room for ATH to satisfy the different optimization requirements of end users.

In summary, our results demonstrate that ATH can improve the performance of HBase workloads significantly as long as the the performance of them is sensitive to the HBase configuration parameters. ATH is a powerful tool to help end users make a good trade-off between throughput and latency.

F. OPTIMUM CONFIGURATIONS

As shown in Table 3, the optimum configurations of the experimented benchmarks are significantly different from the default configuration parameter values. For example, the default value of *io.storefile.bloom.block.size*

TABLE 3. Optimum parameter values for five YCSB workloads when $w_1 = 1.0$ and $w_2 = 0.0$.

Configuration Parameters	Default	update-heavy	read-heavy	read-only	short-ranges	read-modify
hbase.client.max.perregion.tasks	1	5	5	1	3	1
hbase.client.max.perserver.tasks	5	13	19	6	15	8
ycsb.client.threads	64	64	70	65	64	64
hbase.regionserver.handler.count	30	35	60	51	28	68
hbase.ipc.server.callqueue.handler.factor	0.1	0.5	0.6	0.5	0.7	0.5
hbase.ipc.server.callqueue.read.ratio	0.0	0.9	0.9	0.4	0.4	0.9
hbase.ipc.server.callqueue.scan.ratio	0.0	0.1	0.1	0.1	0.9	0.1
regionserver.global.memstore.upperLimit	0.40	0.18	0.52	0.46	0.47	0.55
regionserver.global.memstore.lowerLimit	0.35	0.11	0.47	0.41	0.43	0.50
hbase.hregion.memstore.flush.size	16MB*8	16MB*21	16MB*27	16MB*31	16MB*12	16MB*18
hfile.block.cache.size	0.25	0.50	0.18	0.24	0.22	0.15
hfile.index.block.max.size	64KB*2	64KB*2	64KB*5	64KB*10	64KB*4	64KB*3
hfile.block.index.cacheonwrite	false	true	false	false	false	false
io.storefile.bloom.block.size	64KB*2	64KB*15	64KB*2	64KB*12	64KB*4	64KB*12
hfile.block.bloom.cacheonwrite	false	true	true	true	false	true
hbase.rs.cacheblocksonwrite	false	true	false	true	true	true
hbase.hstore.blockingStoreFiles	7	11	6	11	10	11
hbase.hstore.compactionThreshold	3	6	4	5	5	6
hbase.storescanner.parallel.seek.enable	false	true	true	false	true	true
hbase.storescanner.parallel.seek.threads	5	20	10	7	7	15
hbase.regionserver.hlog.blocksize	16MB*8	16MB*14	16MB*22	16MB*9	16MB*24	16MB*25
hbase.regionserver.maxlogs	32	36	37	20	47	17
java virtual machine heap size	512MB*8	512MB*17	512MB*17	512MB*8	512MB*10	512MB*19

is 64KB*2 while its corresponding values for *update-heavy*, *read-heavy*, *read-only*, *short-ranges*, and *read-modify* are 64KB*15, 64KB*2, 64KB*12, 64KB*4, and 64KB*12, respectively. In addition, *callqueue*-related parameters such as *hbase.ipc.callqueue.handler.factor* and *hbase.ipc.callqueue.read.ratio* are also far from their default values. This indicates that ATH is able to configure a dedicated value of *callqueue* for each type of request for optimized performance.

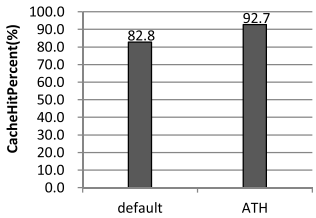

FIGURE 16. Block cache hit rate of *update-heavy* with default and ATH configuration.

G. DETAILED ANALYSIS: UH AND SR

We now further analyze the results of two particular workloads in more details to explain where the throughput improvements come from when ATH is employed. One is *update-heavy*, for which ATH improves its throughput the most significantly. The other is *short-ranges*, for which ATH does not present good scalability. That is: ATH improves the throughput of this workload by 40% when the `target-set` is 0.1 million while the improvement gradually reduces when the size of its `target-set` increases.

Based on our observation, the throughput improvement of *update-heavy* comes from three aspects. (1) By adjusting the values of parameters *hfile.block.cache.size* to 0.5 and *hbase.rs.cacheblocksonwrite* to TRUE, ATH improves the *blockCacheHitPercent* from 82.8% to 92.7% (shown in Fig. 16), which in turn improves the throughput. (2) Rather than keeping parameters *hbase.regionserver.handler.count* and *hbase.ipc.server.callqueue.handler.factor* to their default values (30 and 0.1, respectively), ATH sets the former to 35 and the later to 0.5. The product of these two parameters determines the number of *callqueue*s. In this case, ATH increases this product from 3 to 17, which possibly improves the throughput. (3) ATH also improves the disk I/O throughput by adjusting *hbase.storescanner.parallel.seek.enable* to TRUE and *hbase.storescanner.parallel.seek.threads* to 20. The default values of them are FALSE and 5, respectively, which does not support parallel disk operations.

In addition, the ATH configuration also causes the reduction of execution times of two key stages of a request: (1) a request is waiting in a *callqueue* (the *wrtq* stage); and (2) the request is being executed (the *execute* stage). As shown in Fig. 17, the time used for these two stages with default configuration is 32.8 *ms* whereas that with ATH configuration is 24.7 *ms*, indicating 24.7% of performance improvement.

We now turn to analyze why ATH can not improve the throughput of *short-ranges* when its `target-set` is relatively large. As described in Section IV, *short-ranges* is a network-intensive workload. When the `target-set`

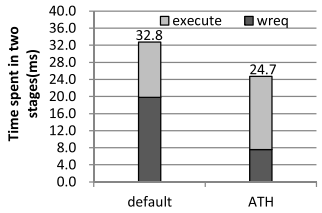

FIGURE 17. Execution times for stages *wreq* and *execute* of *update-heavy*.

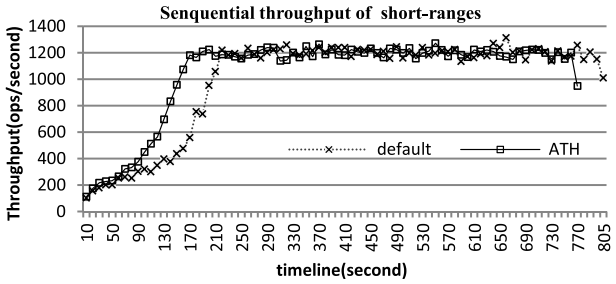

FIGURE 18. The throughput of each time interval for *short-ranges* with the default and ATH configurations. The target-set is 0.8 million, and the time interval is 10 seconds.

TABLE 4. Time cost.

Workload	Profiling(h)	Modeling(s)	Searching(s)
Update-Heavy	23.14	1.37	38.21
Read-Heavy	26.13	0.43	35.69
Read-Only	28.17	0.45	35.59
Short-Ranges	18.25	0.49	34.43
Read-Modify	26.65	0.59	34.14

achieves or exceeds 0.8 million, this workload consumes all the bandwidth between the client and the server. In such a case, No matter how we adjust the configuration parameters, the throughput can not be improved further, as shown in Fig.18. The figure also shows that *short-ranges* with default configuration has the same throughput as it with ATH configuration, which confirms our conclusion.

H. OVERHEAD

We now report the overhead of ATH which includes the times used to collect the training data, to train the performance models, and to search the optimum configurations. Table 4 shows the time costs.

The unit for the time used for collecting data is hour, for model training is second, and for searching optimum configuration is second as well. As can be seen, the highest cost is the time used to collect data and is up to 28.17 hours. While it seems long, it is a one-time cost and is still attractive compared to manual configurations. Moreover, ATH targets to optimize HBase applications which usually run in data centers for months or years. As such, this high one-time cost is amortized between a very large number of runs, leading to a very low cost per run.

The times used to train the performance models for the five benchmarks are less than 2 seconds. This indicates that the time cost for model training is very low. Although the time used to search the optimum configurations by the GA is near 40 seconds, but the time can be ignored when compared to the profiling time.

VI. RELATED WORK

HBase is a popular distributed database system which is widely used in many large- and small-scale internet companies [14]. Since the performance of HBase is poor in many applications, a number of approaches have been proposed from different angles. We classify these approaches into four categories: (1) data related optimization, (2) architecture related optimization, (3) schema related optimization, and (4) configuration parameter tuning.

Data related optimization approaches include *bulk-loading*, *pre-splitting*, *server-side filter*, and *coprocessor* [33]. *bulk-loading* directly transfers the input data into HBase's internal data format. *pre-splitting* splits a heavy workload to many regions served by different regionservers, preventing the requests from flushing into a single region. *server-side filter* facilitates an application to fetch only the relevant data from a HBase table and *coprocessor* sends the business computation codes to the regionserver side. These data related optimization techniques can significantly improve the performance of HBase applications. However, each technique is designed for a particular scenario. It is hard to apply a technique for a scenario to other scenarios. For example, *bulk-loading* is suitable for "*insert heavy*" applications, but it does not work well for "*read heavy*" workloads.

Architecture related optimization tries to optimize the performance of HBase from the architecture angle. Spillane et al. designed a multi-tier storage architecture to replace the original two-layer architecture. This technique adopts modern Flash SSD (Solid-State Disk) as the added storage layer, improving the performance of *MemStore* and *BlockCache* [30]. Harter et al. designed a local-compaction mechanism to replace the original compaction, reducing the amount of data transferred across networks [18].

Schema related optimization is another way to optimize HBase applications. NoSE [28] proposed an approach to guide the mapping from the application's conceptual data model to a database schema. Bermbach *et al.* [7] developed a tool to help design the database schema for a given application. However, it is difficult to employ such techniques for end users without expertise because: (1) the former needs in-depth knowledge about the HBase architecture; and (2) the later needs deep understanding about the data and application.

Configuration parameter tuning is an effective way to improve the performance of HBase applications. The HBase reference guide [33] presents a number of configuration parameter tuning suggestions. For example, it suggests how to adjust the sizes of *BlockCache*, *MemStore*, and RPC *callqueue* to improve the performance of "*read and write*

heavy" applications. Nevertheless, it needs to manually tune the values of these parameters for each HBase application, which is tedious.

PCM [6] is yet another configuration parameter tuning approach. It firstly divides various workloads into many categories according to their read and write ratio. It then suggests manually tuning a group of configurable parameters to find the optimum configuration for each category. Finally, PCM constructs a set of static polices based on previous tuning experiences. When using PCM, one needs to firstly identify an application's characteristics such as write ratio and then finds the matched static policy for performance optimization. ATH differs from PCM in three aspects: (1) PCM is a manual performance tuning approach while ATH is an auto-tuning one; (2) The static policies of PCM can not optimize a wide range of different workloads while ATH can optimize workloads with any read or write ratio; and (3) PCM only takes the throughput as its optimization goal while ATH considers both the throughput and latency as optimization goals.

VII. CONCLUSIONS

In this paper, we propose an approach named ATH to automatically tuning the configuration parameters of HBase for optimized performance. ATH employs random forest to build accurate performance prediction models for throughput and latency for a given HBase workload. It then takes the output of the two models as the input to a genetic algorithm to automatically search the HBase configuration space to finally yield a HBase configuration setting that leads to optimized performance.

We evaluate ATH using five YCSB workloads, each with five target-sets ranging from 1 million to 1200 millions. The results show that ATH speeds up throughput of HBase workloads by 1.41× on average and up to 1.97×. It also reduces the latency by 11% on average and up to 57% at the same time. In addition, we demonstrate that the performance benefits obtained by ATH increase along with the target-set of a HBase workload.

ACKNOWLEDGMENT

The authors would also like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] *Apache Cassandra*, accessed on May 26, 2017. [Online]. Available: http://incubator.apache.org/cassandra/
[2] *Apache CouchDB*, accessed on May 26, 2017. [Online]. Available: http://couchdb.apache.org/
[3] *Apache HBase*, accessed on May 26, 2017. [Online]. Available: http://hadoop.apache.org/hbase/
[4] *HBase at Taobao*, accessed on May 26, 2017. [Online]. Available: http://www.eygle.com/digest/2012/03/hbase-at-taobao.html
[5] *Project Voldemort*, accessed on May 26, 2017. [Online]. Available: http://project-voldemort.com
[6] X. Bao, L. Liu, N. Xiao, Y. Zhou, and Q. Zhang, "Policy-driven configuration management for NoSQL," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 245–252.
[7] D. Bermbach, S. Müller, J. Eberhardt, and S. Tai, "Informed schema design for column store-based database services," in *Proc. IEEE 8th Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, Oct. 2015, pp. 163–172.
[8] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.
[9] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[10] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.
[11] B. Calder *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Oper. Syst. Principles*, Oct. 2011, pp. 143–157.
[12] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," Univ. Southern California, Los Angeles, CA, USA, Tech. Rep. 08-897, 2008, accessed on May 26, 2017. [Online]. Available: https://www.cs.usc.edu/research/technical-reports-list
[13] B. F. Cooper *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *J. Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, New York, NY, USA, 2010, pp. 143–154.
[15] P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia, "Auto-tuning of cloud-based in-memory transactional data grids via machine learning," in *Proc. 2nd Symp. Netw. Cloud Comput. Appl.*, Dec. 2012, pp. 9–16.
[16] B. Efron and R. J. Tibshirani, *An Introduction to Bootstrap*. Boca Raton, FL, USA: CRC Press, 1994.
[17] A. E. Gencer, D. Bindel, E. G. Sirer, and R. van Renesse, "Configuring distributed computations using response surfaces," in *Proc. 16th Annu. Middlew. Conf.*, 2015, pp. 235–246.
[18] T. Harter *et al.*, "Analysis of HDFS under HBase: A facebook messages case study," in *Proc. 12th USENIX Conf. File Storage Technol. (FAST)*, Berkeley, CA, USA, 2014, pp. 199–212.
[19] H. Herodotou. (2011). "Hadoop performance models." [Online]. Available: https://arxiv.org/abs/1106.0940
[20] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.
[21] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *Proc. CIDR*, 2011, vol. 11. no. 2011, pp. 261–272.
[22] T. Katagiri, K. Kise, H. Honda, and T. Yuba, "FIBER: A generalized framework for auto-tuning software," in *Proc. Int. Symp. High Perform. Comput.*, 2003, pp. 146–159.
[23] M. Kumar, M. Husian, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," *Int. J. Inf. Technol. Knowl. Manage.*, vol. 2, no. 2, pp. 451–454, 2010.
[24] P. Lama and X. Zhou, "AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud," in *Proc. 9th Int. Conf. Auto. Comput.*, Sep. 2012, pp. 63–72.
[25] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.
[26] L. Lie, "Heuristic artificial intelligent algorithm for genetic algorithm," *Key Eng. Mater.*, vols. 439–440, pp. 516–521, Jun. 2010.
[27] C. B. Lucasius and G. Kateman, "Understanding and using genetic algorithms part 1. Concepts, properties and context," *Chemometrics Intell. Lab. Syst.*, vol. 19, no. 1, pp. 1–33, May 1993.
[28] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu, "NoSE: Schema design for NoSQL applications," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 181–192.
[29] E. Sciore, "SimpleDB: A simple java-based multiuser syst for teaching database internals," *ACM SIGCSE Bull.*, vol. 39, no. 1, pp. 561–565, 2007.
[30] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, and S. Archak, "An efficient multi-tier tablet server storage architecture," in *Proc. 2nd ACM Symp. Cloud Comput. (SOCC)*, New York, NY, USA, 2011, pp. 1:1–1:14.
[31] T. Tanaka, R. Otsuka, A. Fujii, T. Katagiri, and T. Imamura, "Implementation of D-spline-based incremental performance parameter estimation method with ppOpen-AT," *Sci. Programm.*, vol. 22, no. 4, pp. 299–307, 2014.
[32] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proc. ACM/IEEE Conf. Supercomput.*, 2002, pp. 1–11.
[33] Apache HBase Team. (2016). *Apache HBase Reference Guide*. [Online]. Available: http://hbase.apache.org/book.html
[34] V. Torczon and M. W. Trosset, "From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization," *Comput. Sci. Statist.*, 1998, pp. 396–401.

[35] W. Wang, R.-B. Chen, and C.-L. Hsu, "Using adaptive multi-accurate function evaluations in a surrogate-assisted method for computer experiments," *J. Comput. Appl. Math.*, vol. 235, no. 10, pp. 3151–3162, Mar. 2011.
[36] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 196–205, 2003.

WEN XIONG received the B.S. degree from the Wuhan Institute of Technology in 2005 and the M.S. degree from the HuaZhong University of Science and Technology in 2008. He is currently pursuing the Ph.D. degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research focuses on benchmarking and optimization for big data systems.

ZHENGDONG BEI received the B.S. degree from the National University of Defense Technology in 2006 and the M.S. degree from Central South University in 2009. He is currently pursuing the Ph.D. degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include performance optimization of big data system, data mining, machine learning, and image processing.

CHENGZHONG XU (F'16) received the Ph.D. degree from The University of Hong Kong in 1993. He is currently the Director of the Institute of Advanced Computing and Data Engineering with the Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. His research interest is in parallel and distributed systems and cloud computing. He was a recipient of the Outstanding Overseas Scholar Award of NSFC. He serves on a number of journal editorial boards, including the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the *Journal of Parallel and Distributed Computing*, and the *China Science Information Sciences*.

ZHIBIN YU (M'12) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST) in 2008. He is currently a Professor with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences (CAS). His research interests are computer architecture, workload characterization and generation, GPGPU architecture, and big data processing. He is a member of ACM. He received the Outstanding Technical Talent Program of CAS in 2014 and the "Peacock Talent" Program of Shenzhen City in 2013. He received the first award in teaching contest of HUST young lectures in 2005 and the second award in teaching quality assessment of HUST in 2003. He serves for ISCA 2013, MICRO 2014, ISCA 2015, and HPCA 2015.