OptCon: a Flexible Workload and SLA-Aware Framework for Consistency Tuning

Subhajit Sidhanta
LSU School of Electrical Engineering and Computer Science
ssidha1@tigers.lsu.edu

Wojciech Golab
Department of Electrical and Computer Engineering
University of Waterloo
wgolab@uwaterloo.ca

Saikat Basu
LSU School of Electrical Engineering and Computer Science
sbasu8@tigers.lsu.edu

Supratik Mukhopadhyay
LSU School of Electrical Engineering and Computer Science
supratik@csc.lsu.edu

ABSTRACT

Users of distributed data stores that employ quorum-based replication are burdened with the choice of a suitable client-side consistency setting for each storage operation. The matching choice is difficult to reason about as it depends on the application requirements, as well as the workload, network latencies and processing delays at a given time. We present OptCon: a novel predictive framework that can automate this choice given a user-specified combination of thresholds in the form of an SLA (i.e., service level agreement). While manually tuned consistency settings remain fixed unless explicitly reconfigured by the developer, OptCon consistency levels can change from operation to operation with changing workload and SLA specifications.

OptCon predicts the strongest client-side consistency setting that can be applied under the current workload, network and system state while satisfying the SLA with respect to observed consistency and latency. We show experimentally that OptCon provides an intelligent combination of strong consistency in certain feasible scenarios as well as supporting weaker forms of consistency in other cases based on the demand of the situation. With different SLAs, we demonstrate using the RuBBOS benchmark that OptCon is at least as effective as any optimal combination of manually configured fixed consistency settings in achieving the SLA conditions. Under varying benchmark workloads, OptCon automatically adapts the consistency settings producing staleness and latency that matches the SLA demands.

Categories and Subject Descriptors

2.2 [**Systems Issues for Middleware**]: Consistency, availability, and replication; 1.1 [**Middleware Platforms and Usage Models**]: [Middleware for emerging cloud computing platforms]

General Terms

Management, Performance, Reliability

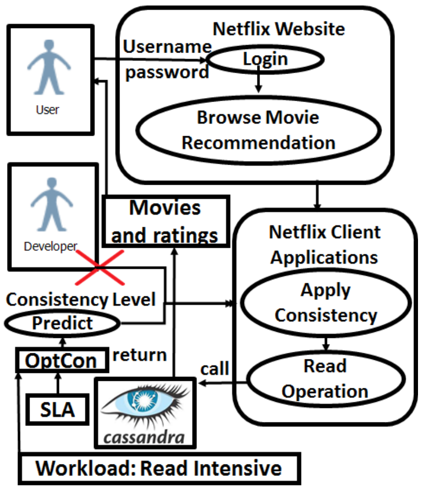
Keywords

Matching Consistency; Consistency-Latency Tradeoff

1. INTRODUCTION

We present OptCon, a novel predictive framework that can automate the tuning of *client-centric* [53] consistency [53] settings for operations performed by client applications on quorum-based distributed data stores. It achieves a combination of user-specified performance thresholds in the form of an SLA. OptCon predicts the strongest client-side consistency setting that can be applied under the given workload, while satisfying the *staleness* [53] and latency bounds in an SLA. Experimental results show that OptCon has less than 15% cross validation error (with Decision Tree) in predicting appropriate consistency settings on an Apache Cassandra 2.1.0 [32] cluster of 20 m3.large Amazon EC2 instances running Ubuntu 13.10 LTS with replication factor of 5. We demonstrate that OptCon is at least as effective as carefully chosen manually configured fixed consistency settings in satisfying latency and staleness thresholds under changing SLAs and with varying workload.

A distributed storage system design must take into consideration two different aspects of consistency [53]: 1) The *data-centric* [53] aspect of consistency considers the service provider's viewpoint, focusing on maintaining the internal state of the system. This is achieved by synchronizing among the nodes in the cluster, preserving the state of the replicas. 2) The *client-centric* [53] viewpoint considers the perspective of the client application, i.e., the consistency observed by the client application. From this viewpoint, the design has to take into account the implications of the consistency settings on the results and performance metrics returned by the client applications to the user. Hence, with client-centric consistency settings the system design must focus on the performance requirements of users given by the SLA guarantees. OptCon addresses the problem of automating the task of tuning the client-centric consistency settings for client applications that access distributed datastores as storage backend, based on application workload and SLA requirements.



(a) Use Case: Browse Movie Recommendation Operation on Netflix. If used with consistency settings predicted by OptCon instead of consistency settings chosen by developers

L	Latency or Time Delay for an Operation
C	The client specified consistency level for both read and write operations
Tc	Thread count, i.e., the number of threads spawned by the client
S	Staleness or Observed Consistency
RW	Read-Write proportion, i.e., the ratio of reads and writes in an operation
P	Packet count , i.e., the number of packets captured while performing the given operation

(b) Glossary

Figure 1: A Use Case of OptCon and a Glossary of Symbols Used

Figure 1(a) demonstrates a use case of operations performed on a distributed datastore with OptCon as an interface for consistency tuning. Netflix [16] stores its data on a Cassandra cluster of 288 m2.xlarge Amazon EC2 instances to support increasing datasize, high availability, and efficient failure recovery. Netflix generally works with read intensive workloads [16] where users browse movie titles, watch movies, or view recommendations. In the given use case (refer Figure 1(a)), the user logs in to the Netflix website using his credentials and explores the browse recommendation operation. This triggers a call to the Netflix client application (see Figure 1(a)). The client applies a client-side consistency setting — either manually chosen by the developer, or given by OptCon. OptCon predicts the suitable consistency settings based on the current workload, and the latency-staleness thresholds specified in the SLA. With the above consistency settings, the client performs a read operation on the Cassandra backend. Cassandra returns the relevant movies and the corresponding ratings, based on the browsing history of the user.

The typical use cases for Cassandra at Netflix [16, 29] require realtime response to user requests. Since eliminating the risk of observing stale data results in higher waiting time owing to synchronization among the replicas [32] — ensuring realtime response entails high chances of getting stale results. Hence, the SLA comprises low latency and higher staleness threshold. Without OptCon, the Netflix application developer manually chooses weak consistency setting ONE READ/QUORUM WRITE under read heavy workload [29]. The chosen consistency setting is applied to the client application which performs the above sequence of operations on Cassandra. If the developer wrongly chooses stronger consistency settings, or if the consistency settings are not correctly modified as per the current workload and SLA, the resultant high latency values would violate the SLA bounds. On the other hand, taking into account read heavy workload and a client thread count of 200 [29], OptCon would predict a matching consistency setting of ONE READ/QUORUM WRITE that would satisfy the given SLA. Otherwise, under write heavy workload, OptCon would apply stronger write consistency settings, i.e., ALL READ/QUORUM WRITE.

2. BACKGROUND AND PRELIMINARIES

Many distributed data storage systems use quorum-based replication to enable fault-tolerance and improve availability [57]. Replication is essential to deal with variations in load, and growing number of parallel requests [23]. Replication comes with the added cost of maintaining consistency across replicas through communication between them [2, 51].

In the early days, distributed storage systems [32, 52, 40, 44, 49, 20] could only be pre-configured with one-time "one size fits all" client-centric consistency settings. However, many distributed data stores [32, 40, 14], nowadays provide the provision of manually tunable consistency options where the users/developers can configure the client-centric consistency: 1) at a data center, 2) for a cluster, or 3) on a per-operation basis. They can programmatically apply the consistency configuration from the client application in the form of a runtime argument, typically referred to as the *consistency level*. A consistency level C is a member of an ordered list C_{list} of categorical attributes. Each element in C_{list} corresponds to: 1) a read consistency level, that corresponds to a specific number of replicas which must respond to a read request from the client before returning, or 2) a write consistency, that corresponds to a specific number of replicas which a write operation must succeed in updating before acknowledging the client.

Distributed data stores like Voldemort [52], Riak [40], and Cassandra [32], offer programmers a host of client-centric consistency level choices, covering a wide range of consistency settings - from eventual to strong consistency. For example, ANY — the lowest read consistency level in Cassandra [32] — requires at least one node from the cluster to respond successfully. The write consistency level QUORUM, placed higher than ANY in C_{list}, requires a quorum (a group) of the replicas to be written. ALL is the highest read consistency level, that requires every replica node in the cluster to successfully respond with results.

To understand the implications of client-side consistency level choices, we consider the case where a client application issues a read request to a Cassandra cluster with replication factor of 5. When issuing a read query [32], the client manually sets the fixed client-side consistency level C to QUORUM via a runtime argument to the read query. Query requests from client applications and synchronization among the replicas are handled by a node called the *coordinator* [32]. Upon receiving the read request, the coordinator obtains the endpoint addresses of all the replica nodes for the given key, and sends out a *data read* request [32] to the nearest replica node (in case of write requests, data writes are sent to all the replicas).

Depending on the consistency level, the coordinator also broadcasts a number of *digest read* commands [32], i.e., probe commands to determine the synchronization among the replicas, to a quorum of replica nodes. Since the quorum comprises three nodes in this case (since replication factor = 5, quorum = (5+1)/2 = 3), three digest read commands are sent out against the requested key. The replica nodes respond to the digest read requests with a hash digest value (i.e., the hash of the actual value) [32], and the timestamp denoting the time of the latest update on that key. Out-of-order values of timestamps in the digest query [32] responses from a replica would imply that the replica is inconsistent. In such cases, a read-repair mechanism [32] runs in the background to bring the stale, i.e., out-of-order, replicas, to a consistent state. However, if the digest read responses are consistent with the data read request (i.e., they have the same timestamp) [32], the coordinator forwards the final result to the client.

The choice of consistency level affects the latency [54] of the operation and the staleness, i.e., the difference in state of the replicas containing the resultant data. For client applications, the latency of responses and the staleness of the resultant data are the most common performance criteria of concern from the perspective of the users [54]. Hence, it is imperative to consider the tradeoff in latency and staleness while choosing the consistency level for a given use case. A single client-centric consistency level may not be appropriate for all use cases [54]. A stringent consistency level, like ALL, may guarantee that a read operation always returns the latest value and ensures that all replicas are updated by a write operation. But it increases the latency for the operation, waiting for responses from all replicas, and may cause indefinite blocking time due to replica outage.

For use cases like status updates [15] , social voting [15], commenting [15], twitter [15], movie recommendation (like the use case in 1(a)), etc., that can tolerate a different view of the data from different clients for a short period of time, eventual consistency might be appropriate as it would result in lower latency [54]. On the other hand, use cases, like customer order [15], bank accounting [15], log management systems [15], etc., working with mission critical data that needs to be correct or complete at all times, require strong consistency while compromising on latency front.

For a real time operation [15] like web search (refer the use case in 1(a)), a user may state that response time for any search query must not exceed 10 milliseconds [54]. In this case, if the user applies a strong consistency level like ALL, a read operation will wait for all the replicas to respond, resulting in latency exceeding the given bound. For banking applications [15] that enable users to check account balance, withdraw or deposit funds, it is crucial that these actions do not leave the database in an incorrect state even for split second. In such cases, the application developer may want to put a low threshold of 5 milliseconds, on staleness of the values being read. Applying weaker consistency settings like ANY will violate the given threshold.

Further, factors like changing workload, network congestion, and node failures may render a particular fixed consistency level infeasible for a given operation at a particular instant of time [36, 54]. Hence, pre-configured or manually tuned consistency settings may not be appropriate for a given operation workload and network state, for the particular latency and staleness requirements demanded by a given use case. The consistency settings needs to be automatically tuned based on the use case, operation type, workload nature and network state.

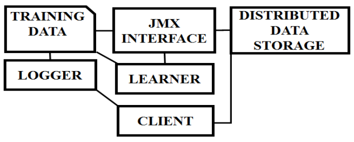
Thus, a particular use case and workload, in a specific network state, demands a very specific consistency level for each operation - which we refer to as the *matching* consistency level. A consistency level is matching with respect to an SLA if the performance metrics—latency and staleness—fall within the thresholds specified in the SLA for an operation performed with that particular consistency level. Hence, to effectively use a Cassandra-like [32] datastore, users need to decide upon the matching consistency level for the given use case and workload. Determining the matching consistency level for an operation necessitates users to have a thorough understanding of: 1) the performance impact of a specific consistency level for a given set of operations, 2) the background synchronization tasks, and 3) the relationship of workload characteristics with the client-side performance.

3. MOTIVATION AND SCOPE

While choosing a matching consistency level from client applications under given SLA, the resultant impact on latency and staleness needs to be considered. For a write operation on a key-value store, the staleness [8] and average latency depends on the following controlling parameters: 1) the client-side consistency level determines the number of replicas to which the data write request needs to be sent [32], 2) the packet count parameter represent the network condition [32], and 3) the read-write proportion and thread count represent the workload characteristics [32]. For a read operation, the staleness and mean latency depends on the same factors as above. For read, the consistency level represents the number of replicas that must respond to digest read requests. The above controlling parameters that, together with consistency level, act as the tuning knob for the observed staleness and latency.

With the current state of the art the developers have to manually determine the matching consistency level for a given operation, taking into consideration the above knob parameters, under the given subSLA thresholds [54]. In the absence of an accurate mathematical model for determining matching consistency level [8] to satisfy staleness and latency thresholds, we use machine learning [22] based prediction techniques [31, 60, 48] to predict the matching consistency level. A model, comprising average latency, staleness, and the knob parameters, can be trained using standard machine learning algorithms, as a one time effort. This model is used for predicting the matching consistency for any given operation, using the knob parameter values and the subSLA thresholds as the test input to the learner (refer Figure 1(a)). We perform a comparative analysis of various machine learning techniques, to provide future researchers with a set of guidelines in applying machine learning for optimization on distributed datastores.

Building a model from training examples is a one time process — the same model can be reused for predicting consistency levels for all operations. Even with failures, the model does not need to be regenerated since the training examples remain unchanged — only the failed predictions need to be rerun.



(a) The Architecture of OptCon

Average Latency	Staleness
≤100ms	≤5ms
≤50ms	≤10ms
≤25ms	≤15ms

(b) Example subSLAs

Figure 2: Architecture and subSLA of OptCon

The threshold values for a given use case are provided to OptCon in form of SLAs from the users. The SLAs are structured as an ordered sequence of subSLA rows similar to the subSLAs of Terry et. al. [54], as illustrated in Figure 2(b). OptCon predicts the matching consistency level under the given input parameters, representing the workload characteristics and nature of the operation.

Application of consistency levels chosen by OptCon, instead of arbitrary or unreliable manually selected consistency levels, ensures that the staleness and average latency bounds in the subSLAs are satisfied. Manually tuned consistency level remains fixed for all operations unless explicitly reconfigured by the developer from the client-side. On the other hand, the consistency level provided by OptCon changes automatically from operation to operation, depending on the workload and subSLA thresholds. Like Cassandra, Voldemort [52] and Riak [40] follow the same Dynamo [20] model. While Cassandra organises the data in form of column families, Voldemort and Riak are strictly key-value stores [32, 52, 40]. Cassandra and Riak are designed to enable faster writes, whereas Voldemort enables faster reads [32, 52, 40]. OptCon can be potentially integrated with any such Dynamo-style system with minor modifications in the wrapper code.

According to the requirements of a particular use case, users can specify different threshold values for the performance metrics in the subSLAs. For read operations, the staleness in the subSLA represents the bound in the timestamp difference between the different versions in the replicas. Higher staleness threshold implies higher read latency overhead owing to waiting for the replicas to respond [8]. For write operations, the staleness threshold represents the time bound that the system must wait before responding to read requests for a written item, in order to allow all the replicas of the said item to commit [8]. The latency threshold corresponds to the blocking time after the write commit. OptCon enables data stores to opportunistically attempt to apply the strongest consistency level that meets the staleness and latency thresholds corresponding to a subSLA. This ensures that the user gets the best possible tradeoff of latency and staleness for a particular use case under a given workload, without violating the thresholds specified in the subSLA.

4. STATE OF THE ART

As it is impossible to achieve all the three objectives of Brewer's CAP theorem (i.e., consistency, availability, and partition-tolerance [11, 24, 3]), large scale distributed storage systems overcome this limitation by offering relaxed consistency guarantees. Storage systems like Google's Spanner [19] are specifically designed to offer strong consistency regardless of the needs of the user and the conditions of the system or the network. Systems like Cassandra, Riak, and DocumentDB [32, 40, 14] allow manual tuning by users/clients to obtain customized consistency guarantees. Recent research has considered techniques for specifying application requirements, including consistency, in a fine-grained manner using service level agreements (SLAs). Terry et. al. [54] introduces Pileus, which can be configured to provide session guarantees such as monotonic reads, as well as bounded staleness. The requirements are specified as a sequence of subSLAs (i.e., rows in an SLA) ordered by a user-defined utility metric, which the system tries to maximize.

Tuba [7], another work by Terry et. al., is a geo-replicated system that supports similar SLAs and is able to automatically reconfigure the configuration settings in the set of replicas in response to changing client locations or request rates. Terry et. al. chooses the best option from SLA rows based on the results obtained from multiple trials, whereas OptCon learns a model and predicts the matching choices, before actually trying out the choices.

Several other systems have been proposed to simplify consistency tuning and could be applied on top of existing eventually consistent data stores. Bailis et. al. [8] uses Monte Carlo methods for calculating the probability of a stale read occurring at a fixed time t following a write. Their mathematical model is based upon the simplifying assumption that writes do not execute concurrently with other operations. For that reason it is not clear how one would compute t from a real workload. Moreover, they do not quantify the effect of client-side consistency levels on latency and staleness. Rahman et. al. [46] provides a probabilistic client-centric consistency-latency trade-off mainly by delaying reads artificially at clients. If a storage system is deployed in a single data center with a fast network then this technique provides a poor trade-off [26]. This is because all reads pay a latency penalty for the sake of a small fraction of reads that would otherwise return a stale value. In comparison, majority quorums can guarantee strong client-centric consistency deterministically as shown in Figure 6(c) of [26].

5. OVERVIEW OF OPTCON

Figure 2(a) describes the architecture of OptCon. It consists of: 1) a Logger module that gathers the performance metrics and system parameters using the Java Management Extensions (JMX) Interface [32] provided by the storage system, 2) a Learner module that runs the Machine Learning on Training Data, and 3) a Client module that predicts the matching consistency level, and performs the requested operation by calling the query client. The JMX Interface module is an extension over YCSB 0.1.4 [18], running on top of a Distributed DataStore [32], collecting various parameters and statistics about the state of the system and network with respect to an operation. The Logger module collates the statistics into a Training Data file.

In the absence of a closed form mathematical model [8] formally relating the parameters—client-side consistency, observed latency, and staleness—we use machine learning to learn a model from historic data. The Learner module builds a classifier from the Training Data, classifying the group of parameters according to the matching client-side consistency. We evaluate various learning algorithms for OptCon using a group of model selection metrics — the developers can use our analysis to decide on the appropriate learning technique to use fro a given distributed datastore. The Client module predicts the strongest consistency level that is possible, under a given set of thresholds specified in the subSLAs. OptCon acts as a wrapper over an existing distributed storage system, allowing users to specify bounds to latency and observed consistency (i.e., staleness) in subSLA format. Though we have experimented with OptCon on Cassandra, it can be potentially integrated with any Dynamo-style [20] distributed data store.

We have chosen the latency L (refer Table 1(b)), and staleness S as the set of performance metrics to be optimized, under a particular set of controlled parameters (tuning knobs): RW, Tc, P, and C (refer Table 1(b)). We use the average latency values measured over a 1 minute time period for L. Staleness S is given by the difference in timestamps of the latest update. It depends on the replication factor and topology, and is measured in terms of the $\mathbf{\Gamma}$ metric [26]. The training data comprises the examples of above variable values collected by the JMX Interface module.

The Γ metric for staleness is based upon Lamport's *atomicity* property [34], which states that operations appear to take effect in some total order that reflects their "happens before" relation in the following sense: if operation A finishes before operation B starts then A must appear to take effect before B. We say that a trace of operations recorded by the logger is $\mathbf{\Gamma}$-atomic if it is atomic in Lamport's sense [34], or becomes atomic after each operation in the trace is transformed by decreasing its starting time by $\mathbf{\Gamma}/2$ time units, and increasing its finish time by $\mathbf{\Gamma}/2$ time units.

In this context the start and finish times are shifted in a mathematical sense for the purpose of analyzing a trace, and do not imply injection of artificial delays; the behavior of the storage system is unaffected. The Γ metric can be defined at various granularity levels, including a *per-key* Γ *score* that quantifies the degree of staleness incurred by operations applied to a particular key [26]. We adopt as our measure of staleness an *average* $\mathbf{\Gamma}$ *score* defined as follows:

$$\frac{\sum \text{per-key } \Gamma \text{ scores}}{\text{total \# of keys accessed in the trace}}$$

6. LEARNING APPROACHES TO PREDICT MATCHING CONSISTENCY

OptCon acts as an interface between client application and the distributed datastore. Hence, speed and simplicity are important criteria for choice of learning techniques in OptCon as we wish to minimize the latency overhead for learning the model. The approach is to favor light weight learning algorithms over compute intensive techniques, while not compromising on accuracy, since compute intensive algorithms generally sacrifices response time to provide higher accuracy. We provide an analysis of the different Learning algorithms and their applicability to OptCon. We leave the choice of suitable learning technique to the developer rendering flexibility to the framework, making it adaptable to other Dynamo-style systems. In the evaluation section (refer section 7), we compare the performance of each of the techniques on Cassandra to enable the developers to make an informed choice.

6.1 Analysis of Learning Approaches Used

We consider Logistic regression [6] and Bayesian learning as [22] they can help visualize the model [22] and provide intuition to the developer. We also consider other approaches — like Decision Tree, Forest, Support Vector Machine, and Neural Network — as they produce better results, being directly computed from the data [22]. We consider these learning approaches in the order of their performance given in Table 1 of Section 7.3.

In the logistic regression [6] approach, we fit two logit (i.e., logistic) functions for the two dependent variables L and S as follows: $logit\,(\pi\,(L)) = \beta_0 + \beta_1 RW + \beta_2 P + \beta_3 Tc$ and $logit\,(\pi\,(S)) = \beta_4 + \beta_5 RW + \beta_6 P + \beta_7 Tc$, where β_i are the coefficients estimated by iterative least square estimation, and π is the likelihood function. Using ordinary least square estimation [22], we iteratively minimize a loss function, given by the difference of the observed and estimated values, with respect to the training dataset. For eliminating overfitting, we use the Lasso [22] method to perform L^1 regularization [6] on the model.

Next we consider the decision tree learning algorithm because of its simplicity, speed, and accuracy. The given problem of choosing the matching consistency level can be viewed as a decision making problem. We use error pruning decision trees, mitigating issues of overfitting [22]. The Decision Tree implementation comprises the following phases: 1) we use exhaustive search [22] for labelling [22] each row in the training dataset with the strongest consistency level C corresponding to a given combination RW, Tc and P, such that the dependent variables are within the respective threshold limits, 2) then we apply Decision Tree learning [22] for training a model from the labelled dataset to predict the matching consistency level.

Next, we consider the random forest [22] algorithm that applies ensemble learning to bootstrap multiple weak decision tree learners to generate a strong classifier. It iteratively chooses random samples with replacement from the training dataset and trains a decision tree on each of the samples. We take the average of the predictions from each decision tree to obtain the final prediction on some test dataset. We introduce low correlation among the individual decision trees by selecting a random subset of the features at each split boundary of the decision trees.

The Bayesian [22] approach makes several simplifying assumptions : 1) we assume a prior logistic distribution for each of RW, Tc, P, and C (see Table 1(b)), 2) we assume that the posterior distributions for L, and S are conditionally dependent on the joint distribution of $\langle RW,\ Tc,\ P,\ C\rangle$, 3) we also assume that the target features L, and S are conditionally independent. With these approximations, we plug in the resultant dependency graph from the Logistic Regression approach as input to the Bayesian Network for generating the Bayesian model.

We also consider Artificial neural networks (ANN) [22] and Support Vector machine (SVM) for learning the model. ANN can provide high accuracy with multiple iterations which may result in high training overhead. SVM can be used to perform multi class classification on nonlinear hyperplane, only if an appropriate kernel function is defined.

7. IMPLEMENTATION AND EVALUATION

We compare the results obtained with various modelling techniques, for the sake of exploring and evaluating the different methods and providing a guideline to future researchers and developers. Our implementation has been built as a Java-based wrapper over the widely used open-source Cassandra datastore v2.1.0 [32]. While performing an operation on the datastore, the users have the choice of: 1) either having OptCon as a layer over the underlying data store, or 2) manually specifying the client-side consistency settings.

7.1 Experimental Setup

We have run our experiments on a test bed consisting of a cluster of 20 Amazon ec2 micro instances, running Cassandra [32], with replication factor of 5, over Ubuntu 13.10 LTS operating system. The source code for OptCon implementation can be found in the github repositories: `https://github.com/ssidhanta/YCSBpatchpredictconsistency/`, `https://github.com/ssidhanta/TrainingModelGenerator/`, and `https://github.com/ssidhanta/HectorCient/`. Without time synchronization among the various nodes, the calculated measures like latency would be rendered useless because of the clock skew. We have used the NTP (Network Time Protocol) protocol [41] implementation from ntp.org to bring the mean clock skew to 1 ms.

7.2 Experimental Procedure

With simulated operations on the above setup, we have generated a dataset of 25,600 rows. For Logistic Regression, SVM, and Neural Network approaches we used Matlab 2013b's statistical toolbox [39]. For the Decision Tree and Random Forest approaches we use Java APIs from Weka [28]. For the Bayesian approach, we use Infer.net [42], an open source machine learning library under the .NET framework. The overhead of OptCon framework, comprises the timeframe starting from the call to the predictor with the input parameters for an operation till the matching predicted consistency level is returned. The average, variance and standard deviation of the OptCon overhead are 2.12 ms, 0.47, and 0.68 respectively.

Studies like [21] suggest that the latency for standard web applications varies in the range of 75-140 ms. The Pileus system [54], which includes latency in subSLAs, uses 200 to 1000 ms as the latency threshold - thus any value between 100 and 1000ms can be a candidate for the threshold of latency in our case. In the work on probabilistically bounded staleness [8], the t-visibility values recorded, in the Riak deployment at Yammer [27], range between 186 and 1364 ms. Hence any value within that range might be used for staleness threshold. We use values in the range 101-150 ms for L and values 41-101 ms for S, for the subSLA thresholds. Default values for the thresholds are: 150 ms for L, and 41 ms for S.

7.3 Comparison of the Models Obtained Using Different Learning Techniques

Approach	Cross Validation Error	AICC	BIC	Overhead
Decision Tree	0.14	10.73	51.44	Low
SVM	0.44	11.21	51.93	Low
Bayesian Learning	0.57	12.85	53.57	Low
Logistic Regression	1.98	16.32	57.07	Low
Random Forest	0.14	9.51	50.24	High
Neural Network	0.059	12.85	63.54	High

Table 1: Model Selection Metrics

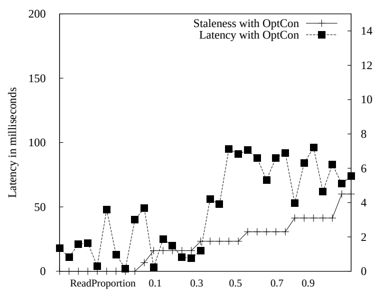
Cross validation (CV) [22] is the most common and widely used model selection technique [22] for comparing models, as it is easy to comprehend and implement. Using 10-fold cross validation, we partitioning the dataset into 10 subsamples, and run 10 rounds of validation. In each round a different subsample is used as the test dataset, while we train the model with the rest of the dataset. We compute the average of the mean squared error values for all the validation rounds to obtain the mean CV error. Table 1 gives the CV error for each of the approaches that OptCon has used.

We also use the Akaike Information Criterion (AIC) [5] which quantifies the information loss during the model training phase. For obtaining AIC, we first compute the likelihood of each of the observed outcomes in the test dataset with respect to the model. We compute the maximum log likelihood L as the maximum of natural logarithms over the likelihood values. AICC [13] further improves upon AIC by penalizing overfitting by introducing a correction term for finite sample size n. Thus, $AICC = 2k - 2\ln L + \frac{2k(k+1)}{n-k-1},$.

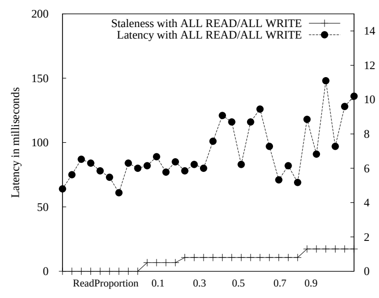
Apart from the AICC, we also compute the Bayesian Information Criteria (BIC) [50] score that uses the Bayesian prior probability estimates. Thus, $BIC = 2k \ln N - 2 \ln L$, where N is the number of examples in the training data. With the additional parameter N for the sample size, BIC assigns a penalty on the sample size than AICC. At smaller sample size, BIC puts lower penalty on the free parameters than AICC, whereas the penalty is higher for larger sample size (because of the term $k \ln N$ instead of k). Table 1 gives the AICC and BIC metrics for each of the OptCon approaches. Normally, the BIC and AICC scores are used together to compare the models — AICC detects problem of overfitting and BIC scores are used to signal the problem of underfitting. Another important criterion for the choice of the algorithm is the learning overhead, which is given in the last column of Table 1.

Logistic regression fairs worst (refer Table 1) among the approaches as indicated by the values of the metrics. This is because it treats the problem as a regression problem [6] and computes a smooth separation plane — whereas our problem is a nonlinear classification problem (linear regression failed in our case). But it produces a simple mathematical model [6] to express the effect of the various controlling parameters RW, Tc, and P, on average latency and staleness for an operation, under different client-side consistency levels C. Also, it can be used to determine the strength of the relationship among the variables. Thus it can act as a basis for further complex modelling algorithms.

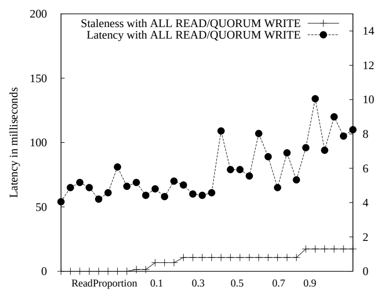
Decision Tree gives very good accuracy and speed (refer Ta-



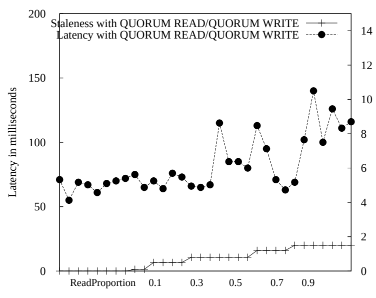
(a) OptCon under subSLA SLA-1 Latency:250ms Staleness: 5ms



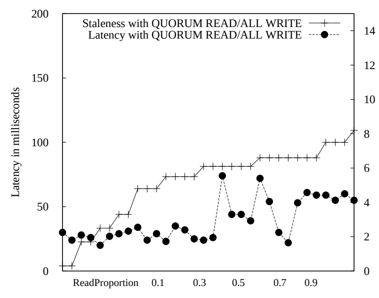
(b) ALL READ/ALL WRITE



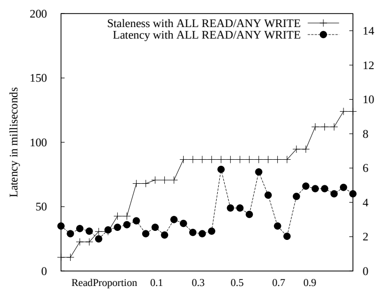
(c) ALL READ/QUORUM WRITE



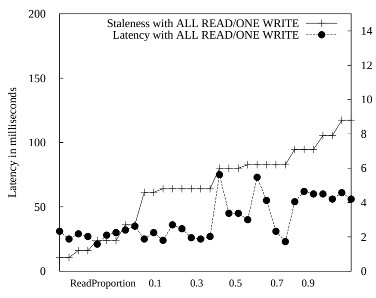
(d) QUORUM READ/QUORUM WRITE



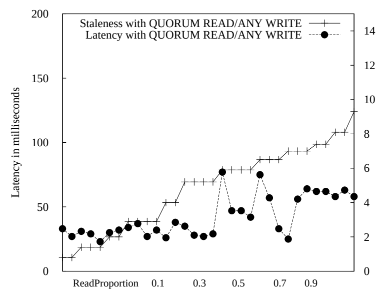
(e) QUORUM READ/ALL WRITE



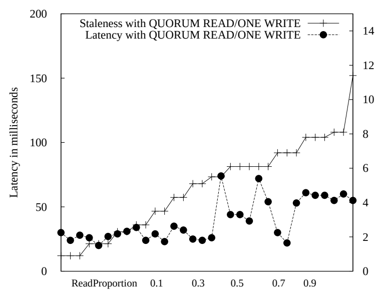
(f) ALL READ/ANY WRITE



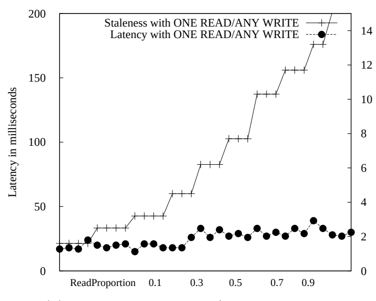
(g) ALL READ/ONE WRITE



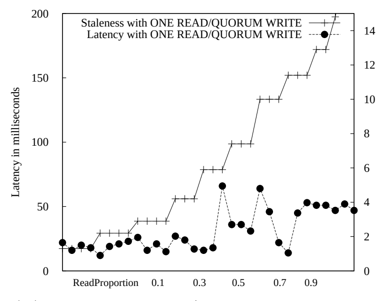
(h) QUORUM READ/ANY WRITE



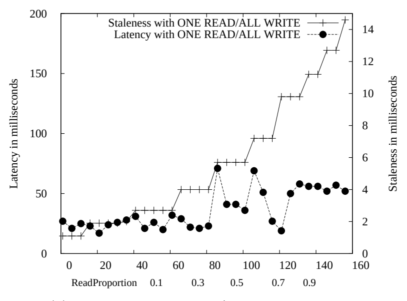
(i) QUORUM READ/ONE WRITE



(j) ONE READ/ANY WRITE



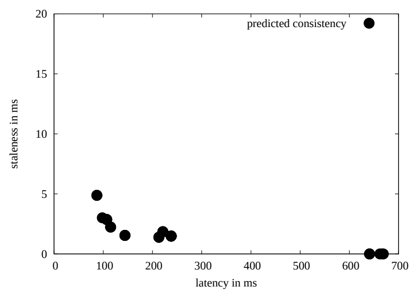
(k) ONE READ/QUORUM WRITE



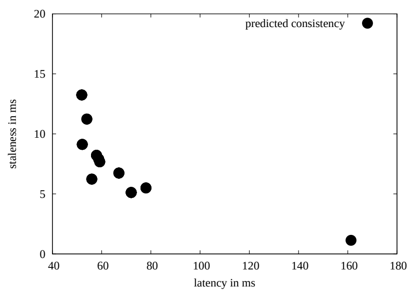
(l) ONE READ/ALL WRITE

Figure 3: Comparison of Latency and Staleness under Varying Read Proportion with OptCon vs Fixed Consistency Settings: Weak Consistency settings ONE/ANY are optimal, i.e., produces acceptable latency and staleness, for low read proportions (see Figures 3(f), 3(g), 3(h), 3(i), 3(j), 3(k), and 3(l)). Strong read-write consistency levels are optimal for high read proportions (see Figures 3(b), 3(c), 3(d), and 3(e)). OptCon is at least as effective as the optimal fixed consistency settings for a particular read proportion under given subSLA (see Figure 3(a)), i.e., it produces at least as low latency and staleness values.

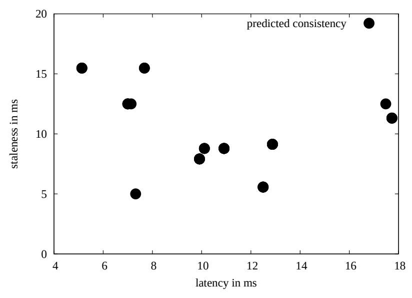
ble 1). This is because the problem can be directly cast as: classifying the training dataset into classes labelled by the strongest feasible consistency levels that result in respective staleness and latency values satisfying the subSLA



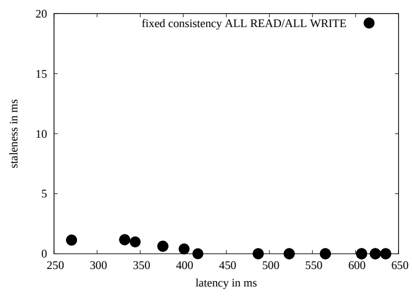
(a) OptCon with subSLA-1: Latency:250ms Staleness: 5ms



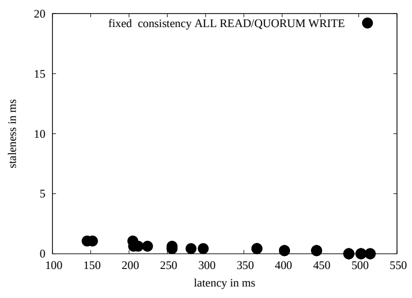
(b) OptCon with subSLA-2: Latency:100ms Staleness: 10ms



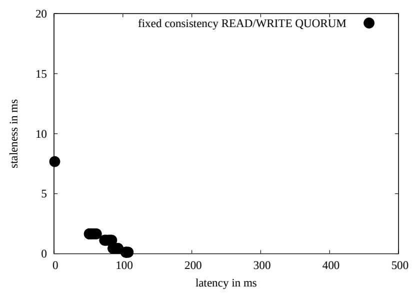
(c) OptCon with subSLA-3: Latency:20ms Staleness: 20ms



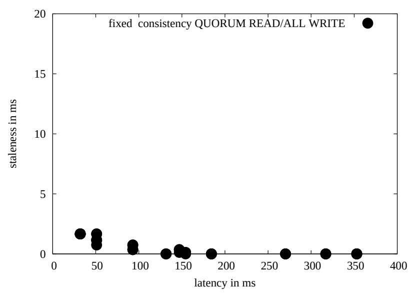
(d) ALL READ/ALL WRITE



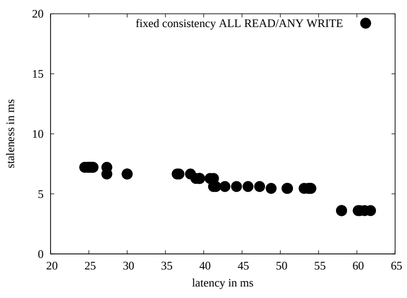
(e) ALL READ/QUORUM WRITE



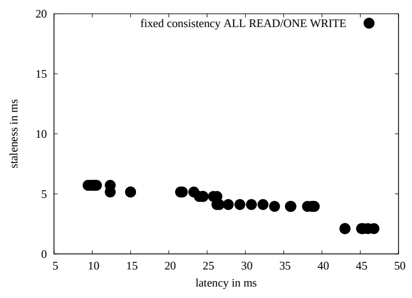
(f) QUORUM READ/QUORUM WRITE



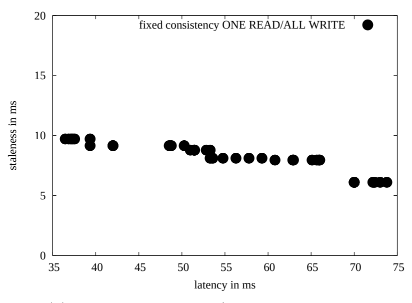
(g) QUORUM READ/ALL WRITE



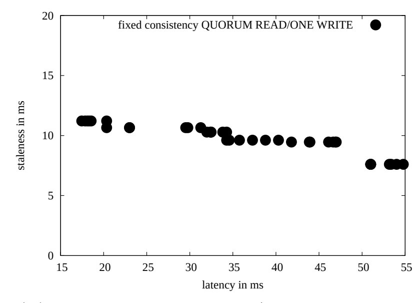
(h) ALL READ/ANY WRITE



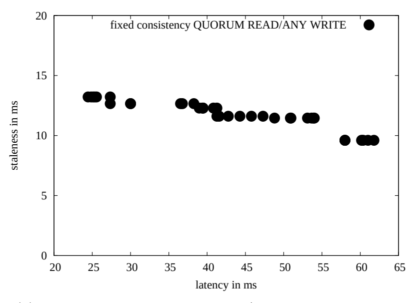
(i) ALL READ/ONE WRITE



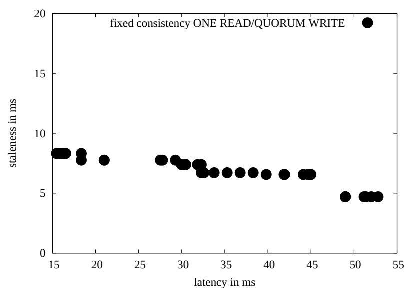
(j) ONE READ/ALL WRITE



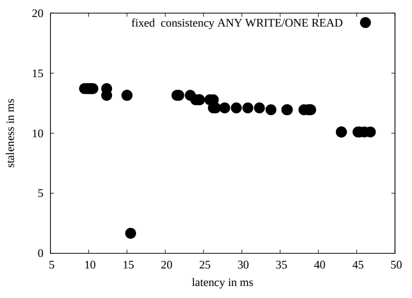
(k) QUORUM READ/ONE WRITE



(l) QUORUM READ/ANY WRITE



(m) ONE READ/QUORUM WRITE



(n) ONE READ/ANY WRITE

subSLA-1	
Consistency	**M Value**
ANY WRITE/ONE READ	68
READ/WRITE QUORUM	41
Predicted Consistency	85
subSLA-2	
Consistency	**M Value**
QUORUM,READ/ALL WRITE	70
Predicted Consistency	92
subSLA-3	
Consistency	**M Value**
ALL,READ/QUORUM WRITE	0
Predicted Consistency	100

(o) Chart Showing M-statistic values with OptCon and Some Fixed Consistency Levels.

Figure 4: Effectiveness of OptCon to adapt to different subSLAs compared to various Fixed Consistency Levels: Strong consistency settings (ALL) are optimal for the subSLA SLA-1 with stringent staleness threshold (see Figures 4(d), 4(e), 4(f), and 4(g)). Weak Consistency levels (ANY/ONE) are optimal for SLA-3 with lower latency threshold(see Figures 4(k), 4(l), 4(m), and 4(n)). Moderate consistency settings are sufficient for SLA-2. OptCon is as effective as the optimal fixed consistency settings for all three subSLAs (see Figures 4(a), 4(b), and 4(c)), i.e., it produces at least as low latency and staleness values.

thresholds under given values for the controlling parameters. Random forest eliminates errors due to overfitting and noisy data by bootstrapping multiple learners [22] coupled with randomized feature selection and produces the best accuracy. However, iterative learning might increase the latency overhead beyond acceptable limit [22].

Though neural network yields high accuracy (Table 1), the iterative learning overhead may well overshoot the threshold for realtime use cases. Bayesian Network requires several approximating assumptions which result in high error scores rendering the approach unfavorable for the given dataset. SVM does not yield acceptable scores because of the problem of using SVM for nonlinear multi class classification. In the absence of a closed form mathematical model for the given problem, we cannot correctly define a suitable kernel function for our nonlinear featureset. The above table can provide as a guide to help developers make informed choice as regards to which learner to implement.

7.4 Adaptability under Varying Read Proportion: YCSB Benchmark Experiments

We run YCSB workloads with varying read proportions on top of the Cassandra test bed. We demonstrate that only a subset of all possible manually chosen read-write consistency levels is *optimal* (i.e., produces acceptable latency and staleness values under a given subSLA) for a specific read proportion. On the other hand, OptCon succeeds in making matching consistency choices for all possible read proportions (refer Figure 3(a)).

As shown by the results in [26], the frequency of stale results (i.e., higher Γ scores) increases with increasing read-proportion in the workload. With higher read proportion, i.e., $RW > 0.5$, stronger consistency settings are required to return less stale data. Also, since the proportion of writes are less, the penalty for the write latency overhead is less - hence stronger consistency settings that result in high latency are acceptable. But, weak consistency settings result in high staleness values for workloads with high read proportion. This results in failure to achieve low staleness thresholds with weaker manually chosen fixed consistency settings as shown in Figures 3(f) through 3(l). Figures 3(b), 3(c), 3(d) and 3(e) demonstrate that the stronger manually chosen read-write consistency levels, i.e., ALL/QUORUM, succeed in lowering staleness values to acceptable bounds under high read proportion.

On the other hand, with lower read proportions the frequency of stale results are smaller [26]. In this case, weaker manually chosen read-write consistency levels, i.e., ONE/ANY (refer Figures 3(f) through 3(l)) are sufficient for returning consistent results. Stronger consistency settings are unnecessary with low read proportions, only resulting in high latency overhead (refer Figures 3(b) through 3(e)). Thus under lower read proportions, weaker manually chosen consistency levels, i.e., ONE/ANY, give better results, i.e., both lower staleness and lower latency, than ALL or QUORUM settings.

Thus, a particular manually chosen fixed consistency setting fails to produce optimal staleness and latency values with different read proportions in the workload — strong consistency settings are suitable for higher read proportions while weaker consistency settings are sufficient for lower read proportions. On the other hand, OptCon can adapt itself with respect to changing workload (see Figure 3(a)), producing matching consistency levels for all possible read proportions.

7.5 Adaptability with varying subSLAs: User Simulation with RUBBoS Benchmark

To study the adaptability of OptCon under various use cases, we have integrated it with the RUBBoS [17] web application benchmark suite. We ran simulations with standard real world user operations like User Registration, Login, Search, Browse, Submission, and Review using RUBBoS on top of the Cassandra cluster with OptCon as a wrapper. We use Decision tree learner as it gives the best combination of CV error, AICC, BIC score, and speed.

The graphs in the Figures 4(a) through 4(n) give comparisons of the observed staleness and latency under different subSLAs: SLA-1, SLA-2 and SLA-3 respectively (refer Table 2(b) and Figures 4(a) though 4(c)) — with OptCon, and with manually chosen fixed consistency levels. To emulate real world applications that work under occasional heavy network traffic, we simulated heavy network congestion scenarios with simulated network delays resulting in arbitrarily high latency values. Such extreme cases (anomalies) of high latency can be observed in a few boundary cases in Figures 4(a), 4(b), and 4(c).

The subSLA SLA-1 represent systems demanding stronger consistency level, such as transaction and batch processing applications. In Figures 4(k) through 4(m) with fixed weak (eventual) read-write consistency settings, i.e, ANY/ONE, the system fails to satisfy the stringent staleness bound in SLA-1. On the other hand, stronger client-side consistency settings (refer Figures 4(d) through 4(g)) produce optimal results — achieving low staleness, while compromising on the latency front. OptCon effectively succeeds in satisfying SLA-1, predicting stronger client-side consistency settings (refer Figures 4(a)) — thus matching the optimal consistency settings for SLA-1.

Higher staleness and lower latency bounds in SLA-3 represents situations where the use case demands weak consistency settings, like real time [54] systems (refer Figure 1(a)) . In this case, the manually chosen strong consistency settings (ALL/QUORUM) unnecessarily produce latencies beyond the acceptable subSLA threshold (refer Figures 4(d) through 4(g)) — whereas weaker (eventual) consistency settings would have been sufficient (refer Figures 4(k) through 4(m)). OptCon, however, achieves the subSLA by predicting weaker consistency settings (refer Figure 4(c)). Similarly with SLA-2, a subset of fixed consistency settings (refer Figures 4(h), 4(i), and 4(j)) produce optimal results, whereas OptCon successfully achieves SLA-2 in Figure 4(b) for all operations.

The effectiveness of OptCon can be evaluated by a metric M, which measures the adaptability [59] [43] of the system under subSLA bounds. The metric M computes the percentage of cases which did not violate the subSLA bounds, given in the Table 4(o). For all the given subSLAs, the M values for operations performed with OptCon, represented by the dots in the Figures 4(a), 4(b), and 4(c), exceed the M values corresponding to operations done with the fixed consistency levels. Thus, it shows that OptCon is at least as effective as an optimal combination of manually chosen fixed consistency settings. On top of that, OptCon succeeds in producing matching consistency choices in cases where fixed consistency levels fail to satisfy the subSLA demands.

The experiments represent the common set of operations for any desktop or web based applications. RUBBoS provides the underlying framework to simulate concurrent access, and interleaving user operations. The above results displays how OptCon can be used effectively to make real world applications work under different latency-consistency subSLAs.

8. RELATED WORK

Eventual consistency was first used in Bayou [55], and later incorporated into a number of storage systems that implement quorum-based replication schemes: Amazon's Dynamo [20, 56], Cassandra [32], Voldemort [52] and Riak [1]. Dynamo, Cassandra and Riak can be configured to provide either eventual consistency or strong consistency using client-side consistency settings. In practice eventual consistency is preferred over strong consistency in scenarios where the system must maintain availability during network partitions, or when the application is latency-sensitive and able to tolerate occasional inconsistencies [3, 11]. Wada et. al. [58] and Bermbach et. al. [9] analyze consistency in commercial cloud storage systems and answered the question "how soon is eventual?" from a system-centric perspective that focuses on the convergence time of the replication protocol. Bailis et. al. [8] and Rahman et. al. [45] instead consider a client-centric perspective in which a consistency anomaly occurs only if differences in state among replicas lead to a client application actually observing a stale read. Staleness can be quantified by defining relaxed consistency properties that generalize Lamport's atomicity property for read/write registers [34]. Aiyer et. al. define k-atomicity, where k is a bound on version-based staleness (i.e., every read returns one of the last k updated values) [4]. Golab et. al. define Δ-atomicity, where Δ is a bound on time-based staleness (i.e., every read returns a value at most Δ time units stale) [25].

The Γ consistency metric, which we use to define the staleness component of an SLA, is a remedy to the problem that Δ is undefined in scenarios where a value appears to be read before it is written due to clock skew [26]. Empirical measurements of Γ and Δ obtained using Cassandra are presented in [26, 45].

A large body of research deals with the problem of supporting various forms of stronger-than-eventual consistency in scalable storage systems and databases. The state machine replication paradigm achieves the strongest possible form of consistency by physically serializing operations [33]. Lamport's Paxos protocol is a quorum-based fault-tolerant protocol for state machine replication [35]. Mencius improves upon Paxos by ensuring better scalability and higher throughput under high client load using a rotating coordinator scheme [38].

A number of scalable fault-tolerant storage systems have been constructed using variations on Paxos: Spinnaker [47], Gaios [10], MDCC [30], and Spanner [19]. These systems use multiple instance of Paxos for scalability, and as a result they incorporate additional mechanisms, such as two-phase commit in Spanner, to support distributed transactions.

Several other papers discuss techniques for supporting transactional semantics at scale, including reducing latency under low contention using "fast" Paxos [30], executing transactions using a combination of eventually consistent and strongly consistent operations [36, 37], providing eventually consistent transactions [12], and using static analysis to prevent conflicts among transactions [62]. Whereas Paxos-based and transactional systems aim to provide strong forms of consistency, our framework is intended to simplify the tuning of systems that support simple read and write operations with weak consistency, for example using sloppy quorums.

Relatively few systems provide mechanisms for fine-grained control over consistency. Yu and Vahdat propose a middleware layer for tunable availability and consistency tradeoffs (TACT) that uses three metrics to express consistency requirements with respect to read and write operations: numerical error, order error, and staleness [61].

9. CONCLUSION

OptCon automates the task of tuning the client-side consistency level in a distributed storage system, which is otherwise manually selected by skilled users, developers or system administrators. This approach enables an intelligent combination of strong consistency, in certain feasible scenarios, and weaker forms of consistency in other cases, depending on the workload and the state of the storage system. We implement the OptCon Learning module using various Learning algorithms and do a comparative analysis of the accuracy and speed for each case (refer Table 1). Our analysis can provide the developers the necessary guidance in making informed choice regarding the suitable learning technique. OptCon has potential in paving the path for future research that may leverage machine learning to perform optimization on distributed storage systems.

Our experiments with the RUBBoS benchamrk show that OptCon is able to predict matching client-side consistency level that satisfies representative SLAs 85% of the time or better by providing optimal combination of latency and staleness. In contrast, a manually chosen client-side consistency level can optimize for either latency or staleness, but not both. We have also demonstrated using varying read proportions in YCSB benchmark workloads, that OptCon can adapt the consistency settings to produce optimal latency and staleness, in contrast to fixed manually chosen consistency settings.

10. REFERENCES

[1] Basho riak. `http://basho.com/riak/`.

[2] M. A. The NoSQL ecosystem. In *In The Architecture of Open Source Applications*, page 185205, 2011.

[3] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb. 2012.

[4] A. Aiyer, L. Alvisi, and R. A. Bazzi. On the

availability of non-strict quorum systems. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 48–62, 2005.

[5] H. Akaike. A new look at the statistical model identification. *IEEE Trans. Automat. Contr.*, 19(6):716–723, Dec. 1974.

[6] G. Andrew and J. Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 33–40, New York, NY, USA, 2007. ACM.

[7] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381, Broomfield, CO, Oct. 2014. USENIX Association.

[8] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, Apr. 2012.

[9] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proc. Workshop on Middleware for Service Oriented Computing (MW4SOC)*, 2011.

[10] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[11] E. A. Brewer. Towards robust distributed systems (Invited Talk). In *Proc. of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2000.

[12] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *Proc. of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 67–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] K. P. Burnham and D. R. Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer Science & Business Media, 2002.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[15] P. Cassandra. Apache cassandra use cases. `http://planetcassandra.org/apache-cassandra-use-cases/`, 2015.

[16] A. Cockcroft and D. Sheahan. Benchmarking cassandra scalability on AWS - over a million writes per second. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`, 2011.

[17] o. Consortium. RUBBoS:˜Bulletin Board Benchmark, 2002.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[20] 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[21] T. Everts. Web performance today.

[22] P. Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012.

[23] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[24] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[25] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *Proc. of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 197–206, New York, NY, USA, 2011. ACM.

[26] W. M. Golab, M. R. Rahman, A. AuYoung, K. Keeton, J. J. Wylie, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ICDCS*, page 28, 2014.

[27] C. Hale and R. Kennedy. Using riak at yammer.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[29] A. Jain. Using cassandra for real-time analytics: Part 1. `http://blog.markedup.com/2013/03/cassandra-real-time-analytics/`, 2011.

[30] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[31] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing Predictive Guarantees for Cloud Networks. In *HotCloud*, „ June 2014.

[32] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS*

Oper. Syst. Rev., 44(2):35–40, Apr. 2010.

[33] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558, 1978.

[34] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 1986.

[35] L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[36] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[37] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[38] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[39] MATLAB. *version 7.10.0 (R2013b)*. The MathWorks Inc., Natick, Massachusetts, 2013.

[40] C. Meiklejohn. Riak PG: Distributed process groups on dynamo-style distributed storage. In *Proc. of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 27–32, New York, NY, USA, 2013. ACM.

[41] D. Mills. Network time protocol (version 3) specification, implementation, 1992.

[42] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[43] A. Paschke and E. Schnappinger-Gerull. A categorization scheme for SLA metrics. *Service Oriented Electronic Commerce*, 80:25–40, 2006.

[44] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[45] M. R. Rahman, W. , A. AuYoung, K. Keeton, and J. J. Wylie. Toward a principled framework for benchmarking consistency. In *Proc. of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 8–8, Berkeley, CA, USA, 2012. USENIX Association.

[46] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya. Probabilistic CAP and timely adaptive key-value stores. 2014.

[47] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243, 2011.

[48] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 85–100, New York, NY, USA, 2013. ACM.

[49] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 41–48, New York, NY, USA, 2008. ACM.

[50] G. Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 6(2):461–464, 1978.

[51] M. Stonebraker. Urban myths about sql, June 2010.

[52] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project Voldemort. In *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[53] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[54] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[55] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182, 1995.

[56] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

[57] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[58] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, pages 134–143, 2011.

[59] Wikipedia. High availability — wikipedia, the free encyclopedia, 2014. [Online; accessed 24-October-2014].

[60] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 123–134, New York, NY, USA, 2013. ACM.

[61] H. Yu and A. Vahdat. Building replicated internet services using TACT: A toolkit for tunable availability and consistency tradeoffs. In *WECWIS*, pages 75–84, 2000.

[62] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.