3-4-2013

Balancing latency and availability in geo-distributed cloud data stores

Shankaranarayanan P N
Purdue University, spuzhava@purdue.edu

Ashiwan Sivakumar
Purdue University, asivakum@purdue.edu

Sanjay Rao
Purdue University, sanjay@purdue.edu

Mohit Tawarmalani
Purdue University, mtawarma@purdue.edu

Balancing latency and availability in geo-distributed cloud data stores

Shankaranarayanan P N

Ashiwan Sivakumar

Sanjay Rao

Mohit Tawarmalsani

TR-ECE-13-03

March 4, 2013

Purdue University

School of Electrical and Computer Engineering

465 Northwestern Avenue

West Lafayette, IN 47907-1285

Balancing latency and availability in geo-distributed cloud data stores

Shankaranarayanan P N
Purdue University
spuzhava@purdue.edu

Ashiwan Sivakumar
Purdue University
asivakum@purdue.edu

Sanjay Rao
Purdue University
sanjay@purdue.edu

Mohit Tawarmalani
Purdue University
mtawarma@purdue.edu

ABSTRACT

Modern web applications face stringent requirements along many dimensions including latency, scalability, and availability. In response, several geo-distributed cloud storage systems have emerged in recent years. Customizing cloud data stores to meet application SLA requirements is a challenge given the scale of applications, and their diverse and dynamic workloads. In this paper, we tackle these challenges in the context of quorum-based systems (e.g. Amazon Dynamo, Cassandra), an important and widely used class of distributed cloud storage systems. We present models that seek to optimize percentiles of response time under normal operation and under a data-center (DC) failure. Our models consider a variety of factors such as the geographic spread of users, DC locations, relative priorities of read and write requests, application consistency requirements and inter-DC communication costs. We evaluate our models using real-world traces of three popular applications: *Twitter*, *Wikipedia* and *Gowalla*, and through experiments with a Cassandra cluster. Our results confirm the importance and effectiveness of our models, and offer important insights on the performance achievable with geo-distributed data stores.

1 Introduction

Modern web applications face stringent performance requirements such as the need to scale to hundreds of thousands of users, achieve consistently low response time for geographically dispersed users, and ensure high availability despite failures of entire DCs (or availability zones). Application latencies and downtime directly impact business revenue [10, 6]– e.g., Amazon found every $100ms$ of latency costs 1% in sales [6]. Further, service level agreements (SLAs) typically require bounds on the $90th$ (and higher) percentile latencies [34, 9].

In response to these challenges, a number of systems that replicate data across geographically distributed data-centers (DCs) have emerged in recent years [25, 37, 34, 26, 24, 40, 17, 7]. Geo-distributing data facilitates service resilience even in the face of wide-area natural disasters, and could lower user perceived latencies by having customers served from DCs close to them.

While geo-distributed cloud data stores offer promise, tailoring these systems to meet the unique requirements of individual applications is a challenge. On the one hand, the performance of these systems critically depends on how many replicas are maintained, which DCs contain what data, as well as the choice of underlying protocol parameters (e.g., quorum sizes in a quorum based system). Choosing an appropriate configuration is challenging since applications are diverse in terms of their workloads (e.g. frequency of reads and writes, and where accesses come from). Applications may also differ in terms of whether SLA requirements are expressed on reads, writes or both. The issues are further exacerbated given the scale of applications (potentially hundreds of thousands of data items), workload diversity across individual data items (e.g. user timelines in Twitter may see very different patterns based on the activity levels and location of a user's friends), and workload dynamics (e.g. due to user mobility, changes in social graph etc.)

In this paper, we present frameworks that can automatically determine how best to customize geo-distributed data stores to meet desired application objectives. Cloud data stores [25, 37, 34, 26, 24, 40] primarily differ in the algorithms used to maintain consistency across distributed replicas, and the consistency guarantees provided. We focus our work on systems such as Amazon's Dynamo [34], and Cassandra [37] that employ quorum protocols – in such systems, reads and writes are simultaneously issued to multiple replicas, and deemed successful only if responses arrive from sufficient replicas to meet a (configurable) quorum requirement. We focus on quorum-based systems given their wide usage in production [34, 37], the rich body of theoretical work they are based on [30, 28, 45, 42], and given the availability of an open-source quorum system [37]. However, we believe our frameworks can be extended to other classes of cloud storage systems as well.

Our frameworks model the response time of cloud storage systems both under normal operation and under the failure of a DC. While quorum protocols have been widely studied in the theoretical distributed systems community [30, 28, 45, 42], our work is distinguished by a focus on new aspects that arise in the context of geo-distributed cloud data stores. In particular, we emphasize percentiles of response time, wide-area latencies, distribution of accesses from multiple geo-

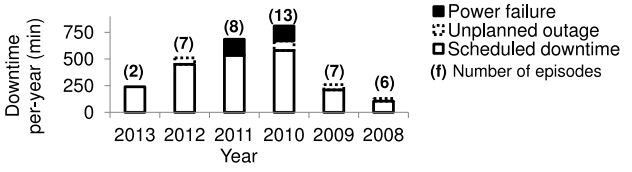


Figure 1: Downtime and number of failure episodes (aggregated per year) of the Google App Engine data store.

graphic regions, and allow for different priorities and delay requirements on read and write traffic. Further, our models consider the impact of DC failures on data store latency, and guide designers towards replica placements that ensure good latencies even under failures. Finally, we consider inter-DC communication costs given this can be an important consideration in cloud deployments.

We validate our models using traces of three popular applications: *Twitter*, *Wikipedia* and *Gowalla*, and through experiments with a Cassandra cluster [37]. While latencies with Cassandra vary widely across different replication configurations, our framework generates configurations which perform within 4% of predicted optimal under realistic experimental settings. Further, our schemes lower latencies during the failure of a DC by as much as 55%, while incurring modest penalties under normal operation. Simulation studies confirm these findings on a larger scale, and highlight the importance of choosing replication configurations differently even across data items of the same application. Overall the results confirm the importance and effectiveness of our frameworks in customizing geo-distributed data stores to meet the unique requirements of cloud applications.

2 Background and Motivation

In this section, we present background on cloud data stores and discuss challenges faced by developers.

2.1 Geo-distributed cloud data stores

In recent years, there has been a growing move towards building geo-distributed cloud storage systems [25, 17, 7, 40, 17]. There are two primary factors that motivate this trend:

• *Availability under DC failures:* Cloud data stores require 5 9's of availability or higher. It is imperative that a data store be capable of handling downtime of an entire DC, which may occur due to planned maintenance (e.g. upgrade of power, cooling and network systems), and unplanned failure (e.g. power outages, and natural disasters) [34, 25, 4, 7]. Figure 1 shows a summary of planned and unplanned failures of the App Engine data store collected from [5].

• *Optimizing client latency:* Interactive web applications (e.g., collaborative editing, Twitter) must meet stringent latency requirements while users are often distributed worldwide (Figure 2). Placing data in DCs closer to users could potentially help. However, the need to maintain consistent application state may limit the benefits of replication as we discuss further in Section 2.3.

A practical and simple approach to geo-replicating data is to use a master-slave system, with master and slave replicas located in different DCs, and data asynchronously copied to the slave [2, 4]. However, slaves may not be completely synchronized with the master when a failure occurs. The system might serve stale data during the failure, and application-level reconciliation may be required once the master recovers. Synchronized master-slave systems on the other hand face higher write latencies. In fact Google App Engine shifted its operations from a master/slave system to a more replicated solution primarily for these reasons [4, 7].

Geo-distributed cloud data stores typically differ in the algorithms used to maintain consistency across replicas, and the consistency semantics provided to applications. Spanner [25] provides database like transaction support by employing 2-phase commit over the Paxos algorithm [38]. Most other systems offer weaker guarantees, primarily with the goal of achieving lower latency as we discuss in the next section. Storage systems also differ in the algorithms used to locate where requested data items are located. Cassandra [37] and Dynamo [34] are key-value stores, and use consistent hashing on the key to identify the nodes on which the data is stored. Spanner maintains explicit directory entries for each group of similarly replicated data items.

2.2 Quorum-based cloud data stores

Quorum protocols have been extensively used in the distributed systems community for managing replicated data [30]. A number of geo-distributed cloud data stores such as Dynamo [34], and Cassandra [37] employ adapted versions of these protocols. In such systems, each data item k may have N_k replicas. All replicas are updated on a write operation, and the write is deemed successful if acknowledgements are received from at least W_k replicas. On a read, the data item is retrieved from all replicas. The read operation terminates when R_k responses are received. In case the replicas do not agree on the value of the data item, typically the most recent value is returned to the user (with ordering established using globally synchronized clocks or causal ordering [16]). The replicas that respond to the reads(writes) constitute the read(write) quorum. The parameters W_k and R_k are referred to as the write and read quorum sizes, and play a critical role in the performance and functioning of the system. Note that quorum sizes may be chosen differently across data items.

In a strict quorum system, the parameters N_k, R_k, and W_k are chosen so as to satisfy the following relationship:

$$R_k + W_k > N_k \tag{1}$$

This ensures that any read and write quorum of a data item intersect, and hence any read operation on a data item will see the result of the last successful write to that item. This property has sometimes been referred to as *strong consistency* [46], although we note that the usage of this term is debatable. Note that consistency guarantees do not hold on operations pertaining to two different data items. Further, quorum data stores do not support database style transactions. Finally, in practice, it is possible to have inconsis-

tencies even on single data items. For e.g., in Cassandra, a failed write may nevertheless update a subset of replicas. We use this terminology in this paper with these caveats.

Dynamo and Cassandra can be configured so the strong consistency requirement is not satisfied. In such a case, read operations can potentially see stale data. In practice, the probability of staleness may be kept low since storage systems attempt to update all replicas on a write operation, and employ background mechanisms to repair stale replicas [16]. data stores configured in this mode are commonly referred to as providing *eventual consistency*, since if no new updates are made to the object, eventually all accesses will return the last updated value. In eventually consistent systems, R_k, W_k and N_k impact the probability of staleness [16].

2.3 Challenges for application developers

While geo-distributing data offers several advantages, deciding how to best replicate data and configure a cloud data store to meet application needs is a challenge for developers for several reasons:

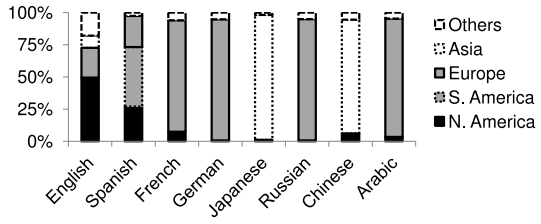


Figure 2: Fraction of accesses from various continents for a few popular languages in the *Wikipedia* application.

• *Scale and workload diversity across data items:* A single application may consist of hundreds of thousands of data items, with individual data items seeing very different workloads. It is desirable to choose replication strategies differently across items. The overheads of maintaining different replication strategies at the granularity of individual items is a potential concern (e.g., directories with meta-data regarding where individual items are located may need to be stored and maintained). However, it is often possible and desirable to choose different replication strategies for various classes of data items. For e.g., Figure 2 shows that the access pattern for *Wikipedia* varies widely across languages, making it desirable to decide replica configuration at the granularity of individual languages. Given the number of classes of data items is still large, determining the appropriate replication strategy for each class is a daunting task for a developer.

• *Minimizing latency while geo-replicating data*: Interactive web applications involve users both reading and writing data (e.g., editing a Facebook wall) and have stronger consistency requirements than traditional content delivery networks (CDNs) where users primarily read data. In CDNs, geo-replication almost always results in latency savings as data is moved closer to users. However, in data stores, these savings must be weighed against the increased latency in maintaining consistency across multiple replicas (e.g. using a quorum protocol). In practice, we are typically restricted to fewer replicas, and the placement of replicas relative to each other is as important as their proximity to users.

• *Naive strategies incur poor performance*: As our evaluations in Sections 8 and 9 will illustrate, the latencies seen with naive replica placement policies is poor, and performance can be significantly improved using more careful placement. Further, even if a placement that ensures good performance under normal conditions is chosen, performance under failure could be poor. Finally, adjusting the read and write quorum sizes can greatly help meet application requirements with different priorities for read and write traffic.

3 System Overview

The data store is deployed in multiple geographically distributed DCs (or availability zones), with each data item replicated in a subset of these DCs. Since our focus is on geo-replication, we consider scenarios where each DC hosts exactly one replica of each item, though our work may be easily extended to allow multiple replicas.

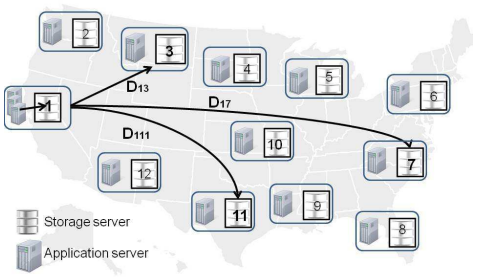


Figure 3: System overview

Applications typically consist of front-end application servers and a back-end layer comprising of multiple storage servers. To read or write data items, an application server contacts a "coordinator" node in the storage service layer. The coordinator determines where the item is replicated (e.g. using consistent hashing or explicit directories), fetches/updates the item using a quorum protocol, and responds to the application server.

We use the term “requests” to denote read/write accesses from application servers to the storage service, and we consider the request to “originate” from the DC where the application server is located. We model “request latency” as the time taken from when an application server issues a read/write request to when it gets a response from the storage service. It is possible that the application issues a single API call to the storage service that accesses multiple data items. (e.g. a multi-get call in Cassandra with multiple keys). We treat such a call as separate requests to each data item.

Users are mapped to application servers in DCs nearest to them through traditional DNS redirection mechanisms [44]. While application servers typically contact a coordinator in the same DC, a coordinator in a near by DC may be contacted if a DC level storage service failure occurs (Section 5).

Table 1: Parameters and inputs to the model

Term	Meaning
M	Number of available DCs.
D_{ij}	Access latency between DCs i and j.
C_i	Cost of outgoing traffic at DC i.
N_i^l	Number of reads/writes from DC i.
T^l	Read/Write Latency Threshold.
p^l	Fraction of requests to be satisfied within T^l.
x_i	Whether DC i hosts a replica.
q_{ij}^l	Whether i's requests use replica in j to meet quorum.
Q^l	Quorum size.
Y_i^l	Whether requests from i are satisfied within T^l.
Y_{ik}^l	Whether requests from i are satisfied within T^l on failure of replica in k.
n_{ij}	Whether reads from i fetch the full data item from j.
l	$l \in r, w$ indicates if term refers to reads/writes.

4 Latency optimized replication

In this section, we present a model that can help application developers optimize the latency seen by their applications with a quorum-based data store. Our overall goal is to determine the replication placement and parameters for each data item. These include (i) the number, and location of DCs in which the data item must be replicated; and (ii) the read and write quorum sizes. We present formulations at the granularity of individual items, however in practice, we expect our formulations to be applied over classes of items that see similar access patterns(Section 2.3).

In this section, we focus on latency under normal operation. In Sections 5 and 6, we show how our models may be extended to consider latency under failure, and incorporate communication costs.

4.1 Minimizing delay of all requests

We begin by discussing a formulation that minimizes the maximum latency across all reads and writes. In Section 4.2, we modify our formulation to bound delays seen by a specified percentage of requests.

We consider settings where the data store is deployed in upto M geographically distributed DCs. D_{ij} denotes the time to transfer a data item from DC j to DC i. For the applications we consider, the size of data items is typically small (e.g., tweets, meta-data, small text files etc.), and hence data transmission times are typically dominated by round trip times.

Let x_i denote an indicator variable which is 1 if DC i holds a replica of the data item and is 0 otherwise. Then, the number of replicas is simply $\sum_j x_j$. Let Q^r and Q^w be the read and write quorum sizes. Then, if the system is configured to meet the strong consistency requirement (Section 2.2), we have:

$$Q^r + Q^w = \sum_j x_j + 1 \tag{2}$$

To model eventually consistent systems, one can instead limit the probability that the application returns stale data to a maximum tolerable threshold specified by the application designer. Recent work [16] has shown how the probability of staleness (e.g. probability a read returns a value older than the last k writes) in a weaker quorum system may be related to the choice of quorum sizes. While we do not further consider such systems in this paper, we note the renewed interest in stronger consistency models in recent years [25, 26, 40].

We next model latencies of individual read and write operations in quorum-based data stores. As discussed in Section 2, even though a read (resp. write) is issued to all replicas, the operation completes from the user perspective when the first Q^r (resp. Q^w) replicas respond. Thus, under normal operation, the latency of read (resp. write) operations issued from DC i is at least the round trip time from i to the Q^r (resp. Q^w) farthest replica closest to DC i. The latency could be higher on failure, which we model in Section 5. Further, specific system implementations may incur higher latencies under certain circumstances. For e.g., on a read operation, the Cassandra system retrieves the full item from only the closest replica, and digests from the others. If a replica besides the closest has a more recent value, additional latency is incurred to fetch the actual item from that replica. We do not model this additional latency since the probability that a digest has the latest value is difficult to estimate and small in practice. Our experimental results in Section 8 demonstrate that, despite this assumption, our models work well in practice.

Let T^r and T^w respectively denote the latency thresholds within which all read and write accesses to the data item must successfully complete. Let q_{ij}^r and q_{ij}^w respectively be indicator variables that are 1 if read and write accesses originating from DC i use a replica in location j to meet their quorum requirements. Then, we have

$$q_{ij}^l \leq x_j \ \forall i, j \ l \in \{r, w\} \tag{3}$$

$$D_{ij} q_{ij}^l \leq T^l \ \forall i, j \ l \in \{r, w\} \tag{4}$$

$$\sum_j q_{ij}^l \geq Q^l \ \forall i \ l \in \{r, w\}. \tag{5}$$

The first two inequalities requires that DC i can use a replica in DC j to meet its quorum only if (i) there exists a replica in DC j; and (ii) DC j is within the desired latency threshold from DC i. The third inequality ensures that, within i's quorum set, there are sufficiently many replicas that meet the above feasibility constraints.

Using the constraints above, a mixed integer programming problem may be formulated to minimize the maximum delay of all reads and writes. We will present a more generalized formulation that minimizes the delay of a desired percentage of requests in the next section.

4.2 Meeting targets on latency percentiles

In practice, it may be difficult to minimize the delay of all requests, and typical Service Level Agreements (SLAs) require bounds on the delays seen by a pre-specified percentage of requests. We now adapt the model of Section 4.1 to this situation.

Let p^r and p^w denote the fraction of read and write requests respectively that must have latencies within the desired thresholds. A key observation is that, given the replica locations, all read and, similarly all write requests, that originate from a given DC encounter the same delay. Thus, it suffices that the model chooses a set of DCs so that the read (resp. write) requests originating at these DCs experience a latency no more than T^r (resp. T^w) and these DCs account for a fraction p^r (resp. p^w) of read (resp. write) requests.

Let N_i^r (resp. N_i^w) denote the number of read (write) requests originating from DC i. Let Y_i^r (resp. Y_i^w) denote indicator variables which are 1 if reads (resp. writes) from DC i meet the delay thresholds, and 0 otherwise. Then, we have:

$$\sum_i N_i^l Y_i^l \geq p^l \sum_i N_i^l \quad \forall i;\ l \in \{r, w\} \tag{6}$$

Further, we relax (5) since only requests originating from a subset of DCs with Y_i^l must meet the delay threshold. Thus, we have:

$$\sum_j q_{ij}^l \geq Q^l Y_i^l \quad \forall i;\ l \in \{r, w\} \tag{7}$$

While (7) is not linear, it may be replaced with the following linear constraints [33]:

$$\sum_j q_{ij}^l \geq Q^l + M Y_i^l - M \ \forall i\ l \in \{r, w\} \tag{8}$$

$$\sum_j q_{ij}^l \geq Y_i^l \ \forall i\ l \in \{r, w\} \tag{9}$$

Note that M is an upper bound on Q^r and Q^w. It is easy to verify that (8) and (9) are equivalent to (7) when Y_i^l is either 0 or 1.

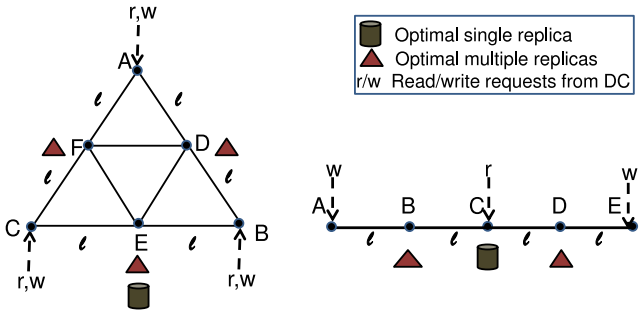
To determine the lowest latency threshold for which a feasible placement exists, we treat T^r and T^w as variables of optimization, and minimize the maximum of the two variables. We allow weights a^r and a^w on read and write delay thresholds to enable an application designer to prioritize reads over writes (or vice-versa). In summary, we have the ***Latency Only*(LAT)** model:

(LAT) min T
subject to $T \geq a^l T^l, \quad l \in \{r, w\}$
Consistency constraint (2)
Quorum constraints (3), (4), (8), (9)
Percentile constraints (6)
$Q^l \in \mathbb{Z}, \quad l \in \{r, w\}$
$q_{ij}^l, x_j, Y_i^l \in \{0, 1\}, \quad \forall i, j; l \in \{r, w\}$

When $p^r = p^l = 1$, (LAT) minimizes the delay of all requests and we refer to this special case as (LATM).

We close this section by discussing straight-forward variations of (LAT) that may be of practical interest. First, while (LAT) explicitly requires a fraction of reads, and a fraction of writes to be satisfied, it is easy to capture a requirement that a fraction p_t of all requests (whether read or write) meet the delay thresholds. This is achieved by simply replacing (6) with the constraint: $\sum_i \sum_l N_i^l Y_i^l \leq p_t \sum_i \sum_l N_i^l$.

Second, it may be desirable to identify the maximum percentage of requests that could meet a specified delay threshold for any data item. This is easily achieved by modifying (LAT) and making the fractions p^r and p^w variables, replacing variables T^r and T^w by a constant threshold, and modifying the objective function to maximizing the minimum of p^r and p^w. Finally, it may be desirable to identify the maximum percentage of requests across *all data items* that could meet a specified delay threshold. This is achieved by maximizing the percentage of requests that could meet the threshold for each individual key and composing the results.



(a) Symmetric spread (b) Asymmetric spread

Figure 4: (a) An optimal multi replica solution with $Q^r = 2, Q^w = 2$ ensures a latency threshold of l, while an optimal single replica solution increases it to $\sqrt{3}l$ (b) An optimal multi replica solution with $Q^r = 2$, $Q^w = 1$ ensures a latency threshold of l, while an optimal single replica solution constrains the delay to $2l$

4.3 How much can replication lower latency?

Given the consistency requirement of quorum data stores, can replication lower latency, and, if so, by how much? In this section, we present examples to show that replication can lower latency, and provide bounds on the *replication benefit* (ratio of optimal latency without and with replication). In assessing the benefits of replication, two key factors are (i) *symmetric/asymmetric spread:* whether read and write requests originate from an identical or different set of DCs; and (ii) *symmetric/asymmetric weights:* whether the weights attached to read and write latency thresholds (a^r, a^w) are identical or different.

Figure 4(a) presents an example where spread and weights are symmetric and the *replication benefit* is $\sqrt{3} \approx 1.732$. When replicas can be placed arbitrarily on a Euclidean plane, it can be shown via an application of Helly's theorem [19] that the *replication benefit* is bounded by $\frac{2}{\sqrt{3}} \approx 1.155$. The setup of Figure 4(a) shows that this is a tight bound since replication achieves this benefit over single placement at the centroid of the triangle. If reads and writes arise from a different set of DCs, replication presents new opportunities to improve latency thresholds even when weights are symmetric. Figure 4(b) illustrates this, and shows the asymmetric spread of reads and writes could be exploited to achieve a replication benefit of 2. The replication benefit can be even higher if asymmetric weights are used. More generally, we

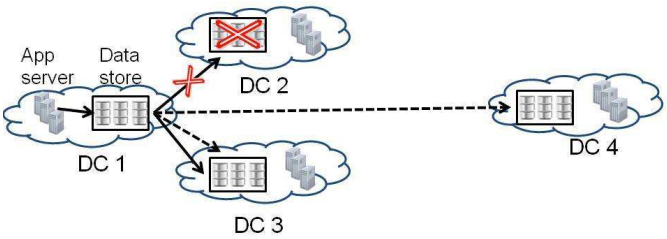


Figure 5: Failures may impact latencies.

have the following observations:

OBSERVATION 1. *With symmetric spread and metric delays, the replication benefit for (LATM) is at most* $2\frac{\max(a^r,a^w)}{\min(a^r,a^w)}$.

Let T_s^l (resp. T_m^l) denote the read latency when $l = r$ and write latency when $l = w$ for a single replica (resp. multiple replicas). Let D_m denote the maximum distance between all pairs of DCs with read/write accesses. Then, it is easy to see that $T_s^l \leq D_m$. Further, since read and write quorums intersect and triangle inequality holds, it follows that $\max\{T_m^r, T_m^w\} \geq \frac{1}{2}(T_m^r + T_m^w) \geq \frac{D_m}{2}$. Observation 1 follows because $\max\{a^r T_s^r, a^w T_s^w\} \leq \max\{a^r, a^w\} D_m$ and $\max\{a^r T_m^r, a^w T_s^w\} \geq \min\{a^r, a^w\}\frac{D_m}{2}$.

OBSERVATION 2. *With asymmetric spreads and metric delays, the replication benefit for (LATM) and (LAT) is at most* $4\frac{\max(a^r,a^w)}{\min(a^r,a^w)}$.

It suffices to show the result for (LAT) since it generalizes (LATM). Let M_r (resp. M_w) be the set of data centers whose read (resp. write) requests are satisfied within the optimal latency. Define $D_{l_1 l_2} = \max_{m_1 \in M_{l_1}} \max_{m_2 \in M_{l_2}} D_{m_1 m_2}$, where $l_1, l_2 \in \{r, w\}$. Then, by placing a replica at some $s \in M_r$ and using triangle inequality, it follows that $T_s^l \leq \max\{D_{rr}, D_{rw}\} \leq 2D_{rw}$. Using the inequality derived for Observation 1, $\max\{a^r T_m^r, a^w T_m^w\} \geq \min\{a^r, a^w\}\frac{D_{rw}}{2}$. Therefore, Observation 2 follows.

5 Achieving latency SLAs despite failures

So far, we have focused on replication strategies that can optimize latency under normal conditions. In this section we discuss failures that may impact entire DCs, and present strategies resilient to them.

5.1 DC wide failures

While several widely deployed techniques exist to protect against individual failures in a DC [27], geo-distributed DCs are primarily motivated by failures that impact entire DCs, or the storage service of an entire DC. While recent works have characterized storage and network failures within a DC [27, 32], there are no publicly available statistics on failures across DCs to the best of our knowledge. Anecdotal evidence, and discussions with practitioners suggests that while DC level failures are not uncommon (Figure 1), it is relatively rare (though certainly feasible) to see correlated failures of multiple geographically distributed DCs. Operators strive to minimize simultaneous downtime of multiple DCs through careful scheduling of maintenance periods, and gradual roll-out of software upgrades.

If cloud data stores employ the strict quorum requirement, it is impossible to guarantee availability with network partition tolerance [31]. In practice, typical network outages partition one DC from others and only impact requests that originate in that DC. More complex partitions, though less frequent, could cause a larger disruption. While not our focus here, it may be an interesting study to explore the trade-off between weaker quorum requirements [16] and the ability to handle complex network partitions.

5.2 Failure resilient replication strategies

While a sufficiently replicated geo-distributed cloud data store may be available despite a DC failure, the latency are likely negatively impacted (Figure 5). We present replication strategies that are resilient to such failures. Pragmatically, we first focus on single DC failures. Then, in Section 5.3, we show how our models easily extend to more complex failure modes. Our models are:

Basic Availability Model (BA): This model simply optimizes latency using formulation (LAT) with the additional constraints that the read and write quorum sizes are at least 2 (and hence the number of replicas is at least 3). Clearly, read and write requests can still achieve quorum when one DC is down and basic availability is maintained. This model does not explicitly consider latency under failure and our evaluations in Section 8 indicate that the scheme may perform poorly under failures – for e.g., the 90^{th} percentile request latency for English *Wikipedia* documents increased from 200 ms to 275 ms when one replica was unavailable.

N-1 Contingency Model (N-1C): This model minimizes the maximum latency across a pre-specified percentile of reads and writes allowing at most one DC to be unavailable at any given time. The model is motivated by contingency analysis techniques commonly employed in power transmission systems [36] to assess the ability of a grid to withstand a single component failure. Although this model is similar in structure to (LAT), there are two important distinctions. First, the quorum requirements must be met not just under normal conditions, but under all possible single DC failures. Second, the desired fraction of requests serviced within a latency threshold, could be met by considering requests from different DCs under different failure scenarios.

Formally, let p_f^r (resp. p_f^w) be the fraction of reads (resp. writes) that must meet the delay thresholds when a replica in any DC is unavailable. Note that the SLA requirement on failures may be more relaxed, possibly requiring a smaller fraction of requests to meet a delay threshold. Let Y_{ik}^r (resp. Y_{ik}^w) be indicator variables that are 1 if read (resp. write) requests from DC i are served within the latency threshold when the replica in DC k is unavailable. Then, we re-

place (6) and (7) with the following:

$$\sum_i Q_i^l Y_{ik}^l \geq p_f^l \sum_i N_i^l \ \forall i \forall k \quad (10)$$

$$\sum_{j,j\neq k} q_{ij}^l \geq Q^l Y_{ik}^l \ \forall i,k \ \ l \in \{r,w\} \quad (11)$$

The first constraint ensures that sufficient requests are serviced within the latency threshold no matter which DC fails. The index k for the Y variables allows the set of requests satisfied within the latency threshold to depend on the DC that fails. The second constraint ensures that the quorum requirements are met when DC k fails with the caveat that DC k cannot be used to meet quorum requirements. At first glance, one may be tempted to introduce an index k for the q variables. However, our computational experience found that the resulting formulation is large and difficult to solve. Instead the more compact and tighter formulation presented was obtained by exploiting the following fact. If a DC j is in the read (resp. write) quorum for a request from i when another DC k, $k \neq j$, fails then the read (resp. write) latency is bounded from below by d_{ij}. Then, whenever a third DC k', $k' \notin \{j,k\}$ fails, using j for read (resp. write) quorum imposes no extra penalty in latency. Therefore, q_{ij}^r (resp. q_{ij}^w) variables can be used to model that DC j is used to satisfy the read (resp. write) quorum when some DC other than j fails. We remark that (11) may be linearized in a manner similar to (7). Putting everything together, we have:

(N-1C) min T_f

subject to
$T_f \geq a^l T^l, \quad l \in \{r,w\}$
Consistency constraint (2)
Quorum constraints (3), (4), (11)
Percentile constraints (10)
$Q^l \in \mathbb{Z}, \quad l \in \{r,w\}$
$q_{ij}^l, x_j, Y_{ik}^l \in \{0,1\}, \forall i,j,k; l \in \{r,w\}.$

For the English *Wikipedia* example, using (N-1C) only increases the delay to 210 ms when one replica is unavailable.

5.3 Model Enhancements

We discuss enhancements to the *N-1 Contingency* model:

Modeling redirection penalties: The model, as presented, neglects that when a replica in DC i is unavailable, requests originating in i may need to be redirected to another DC and thereby incur additional latency. In practice, the impact of such redirection is expected to be modest since there are typically multiple close-by DCs in any geographic region. Further, the redirection penalty only impacts the requests that originate at the failed DC. That said, it is easy to extend the model to consider redirection latency as follows. Replace each indicator variable q_{ij}^l with two sets of variables which respectively denote whether i uses j to meet its quorum requirements when (i) DC i fails (s_{ij}^l); and (ii) any DC other than i fails (t_{ij}^l). Quorum constraints are expressed in similar fashion to (N-1C) for each of these variables. Then, the quorum delay constraint (4) is replaced with two constraints, $(D_{in} + D_{nj})s_{ij}^l \leq T^l$ and $D_{ij}t_{ij}^l \leq T^l$, where n is the DC closest to i. The first constraint models that if DC i fails, requests from i are redirected to n.

Jointly considering normal operation and failures: Formulation (N-1C) finds replication strategies that reduce latency under failure. In practice, a designer prefers strategies that work well in normal conditions as well as under failure. This is achieved by combining the constraints in (LAT) and (N-1C), with an objective function that is a weighted sum of latency under normal conditions T and under failures T_f. The weights are chosen to capture the desired preferences.

Failures of multiple DCs: While we expect simultaneous failures of multiple DCs to be relatively uncommon, it is easy to extend the formulation above to consider such scenarios. Let K be a set whose each element is a set of indices of DCs which may fail simultaneously and the application designer is interested in guarding the application performance against such a failure. We then employ (N-1C) but with k iterating over elements of K instead of the set of DCs. A naive approach may exhaustively enumerate all possible combination of DC failures, could be computationally expensive, and may result in schemes optimized for unlikely events at the expense of more typical occurrences. A more practical approach would involve explicit operator specifications of correlated failure scenarios of interest. For e.g., DCs that share the same network PoP are more likely to fail together, and thus of practical interest to operators.

6 Cost-sensitive replication

When geo-distributed datastores are deployed on public clouds [8]), it may be important to consider dollar costs in addition to latency and availability. In this section, we show how to extend our models to take costs into consideration.

Since our focus on this paper is on geo-distribution, we focus on costs associated with wide-area communication. The storage and computation costs are likely comparable to a data store deployed in a single DC with equivalent number of replicas per data item.

Most cloud providers today charge for out-bound bandwidth transfers at a flat rate per byte (in-bound transfers are typically free), though the rate itself depends on the location of the DC. Let C_i be the cost per byte of out-bound bandwidth transfer from DC i. Using standard modeling techniques, our models can be extended to tiered pricing schemes where the costs per byte differs based on the total monthly consumption.

We next model the communication costs associated with a quorum system. The communication costs depend on control messages employed by the system (e.g., Cassandra uses Gossip messages to ensure nodes can keep track of other live nodes), as well as costs associated with writing and retrieving data items. We do not consider control traffic since it is typically small compared to the data traffic.

Consider an operation that originates in DC i and involves writing a data item whose size is S bytes. Since all replicas must be updated the total cost of a single write operation

is $SC_i \sum_j X_j$ (recall from Section 4.1 that the number of replicas is simply $\sum_j X_j$). Then, the total cost associated with all write operations is:

$$\text{Write Cost} = \sum_i N_i^w SC_i \sum_j X_j \quad (12)$$

In principle, read operations retrieve the data item from all replicas. In practice, systems may employ heuristics to minimize the overhead. For e.g., in typical operation, Cassandra fetches the full data item only from one replica and digests from others. It is possible that the system is in an inconsistent state, and a replica sending a digest has a newer value than the one sending the data item. In such a case, the system must retrieve the full data item from additional replicas. We model the common case where an item is fetched from a single replica. We note that it is difficult to obtain analytical expressions for the probability that more than one replica is requested to send the full data item and that, in practice, this probability is typically low.

Let n_{ij} denote an indicator variable, which is 1 if the full data item is fetched from DC j. The size of the digest is assumed negligibly small. Then, the cost associated with a single read operation originating from i is $S \sum_j n_{ij} C_j$. The total cost associated with all read operations is:

$$\text{Read Cost} = \sum_i \sum_j N_i^r n_{ij} SC_j \quad (13)$$

Putting everything together, we have:

$$\begin{array}{lll} \text{(COST)} & \min & \text{Write Cost + Read Cost} \\ & \text{subject to} & T \geq a^l T^l, \quad l \in \{r, w\} \\ & & n_{ij} \leq q_{ij}^r \\ & & \sum_j n_{ij} = 1 \\ & & \text{Consistency constraint (2)} \\ & & \text{Quorum constraints (3), (4), (8), (9)} \\ & & \text{Percentile constraints (6)} \\ & & Q^l \in \mathbb{Z}, \quad l \in \{r, w\} \\ & & q_{ij}^l, x_j, Y_i^l \in \{0, 1\}, \quad \forall i, j; l \in \{r, w\} \end{array}$$

Note that (COST) is a slight variant of (LAT). A key difference is that threshold T is a fixed parameter rather than a variable of optimization, and (COST) seeks to minimize costs given a delay threshold. Further, two constraints on n_{ij} are added. These require that read requests originating from DC i fetch the entire data item from exactly one DC. Further, the data item must be fetched from a replica that meets the quorum constraints for reads originating from DC i. It is straight-forward to combine (COST) and (N-1C) to present a formulation that simultaneously considers latency, costs, and availability. We do not present this extension for ease of exposition.

7 Evaluation Methodology

We discuss our approach to evaluating our various replication strategies for geo-distributed cloud data stores: *Latency Only* (LAT), *Basic Availability* (*BA*), and *N-1 Contingency* (*N-1C*). Recall that (i) LAT minimizes latency for a desired percentage of requests but does not guarantee availability on failure; (ii) *BA* optimizes latency under normal operations while guaranteeing the system is available on a single DC failure; and (iii) *N-1C* optimizes the latency explicitly for a single DC failure in addition to guaranteeing that the system is available.

Table 2: Trace characteristics

Application	# of keys/classes	Span
Twitter[39]	3,000,000	2006-2011
Wikipedia[13]	196[1]	2009-2012
Gowalla[23]	196,591	Feb 2009-Oct 2010

Our evaluations explore several questions:

- How effectively do our models predict performance in practice?
- How significant are the benefits of our strategies compared to naive off-the-shelf approaches used in current data stores?
- How sensitive are our strategies to application workloads and model parameter choices ?
- Does replication lower latency under normal operations?
- Is it important for a replication strategy to optimize the latency under failures, and what is the penalty ?

We explore these questions using trace-driven simulations from three real-world applications: *Twitter*, *Wikipedia* and *Gowalla*, as well as experiments on a Cassandra cluster deployed on an Amazon EC2 test-bed. Test-bed experiments enable us to validate our models, and to evaluate the benefits of our approach in practice. Simulation studies enable us to evaluate our strategies on a larger scale (hundreds of thousands of data items), and to explore the impact of workload characteristics and model parameters on performance.

We implemented our models using GAMS [20] (a modeling system for optimization problems) and solved them using the CPLEX optimizer. On average, we find that the LAT, *BA*, and *N-1C* models solve within 0.16, 0.17 and 0.41 seconds respectively on a 4 core, $3GHz$, $8GB$ RAM machine. Note that the *N-1C* model takes slightly longer to solve than the other two models as it introduces an extra dimension(k) to certain variables in order to account for the non-availability of a DC. We believe this performance is quite acceptable in practice because we expect our computations to run offline, and on groups of related keys.

7.1 Application workloads

The applications we choose are widely used, have geographically dispersed users who edit and read data, and fit naturally into a key-value model. We note that both *Twitter* and *Gowalla* are already known to use Cassandra [12]. We discuss details of the traces below (see 2 for a summary):

Twitter: We obtained *Twitter* traces [39] which included a user friendship graph, a list of user locations, and public tweets sent by users (along with timestamp information) over a 5 year period. We analyzed Twissandra, an open-source

[1]Aggregating all articles per language (e.g. 4 million articles in English *Wikipedia* are aggregated.)

twitter-like application, and found three types of data items: users, tweets and timelines. We focus our evaluations on timeline objects which map each user to a list of tweets sent by the user and her friends. Writes to a timeline occur when the associated user or her friends post a tweet, and can be obtained directly from the trace. Since the traces do not have read information, we model reads by assuming each user reads her own timeline periodically (once every 10 minutes), and reads her friend's timeline with some probability (0.1) each time the friend posts a tweet.

Wikipedia: We obtained statistics regarding *Wikipedia* usage from [13], which lists the total number of views and edits, as well as the breakdown of user views and edits by geographic region for each language and collaborative project. The data spans a 3 year period with trends shown on quarterly basis. Our model for the *Wikipedia* application consists of article objects with the document id as a key and the content along with its meta data (timestamps, version information, etc). Article page views are modeled as the reads while page edits are modeled as writes. Since per article access data is not available, we model all articles of the same language and project as seeing similar access patterns. We believe this is reasonable since the access pattern is likely dominated by the location of native speakers of each language.

Gowalla: *Gowalla* is a (now disabled) geo-social networking application where users "check-in" at various locations they visit and friends receive all their check-in messages. The traces [11] contained user friendship relations, and a list of all check-ins sent over a two year period. Since the application workflows are similar, we model *Gowalla* in a similar fashion to *Twitter*. Check-ins represent writes to user timelines from the location of the check-in, and reads to timelines were modeled like with *Twitter*.

7.2 DC locations and user distribution

In all our experiments, we assume the data store cluster to comprise of nodes from each of 27 distinct DCs, the locations of which were obtained from AWS Global Infrastructure [1]. Of these DCs, 13 were located in the United States, 8 in Europe, 4 in Asia, 1 in South America and 1 in Australia. We determined the latency between each pair of DCs through delay measurements between Planet-lab nodes that were close to each location. Delay measurements were collected simultaneously between all pairs of locations over a few hours, and median delays were considered. We mapped users from their locations to the nearest DC. Since the locations are free-text fields in our traces, we make use of geocoding services [3] to obtain the user's geographical coordinates.

Our test-bed set up consists of a Cassandra cluster with 27 nodes (corresponding to the 27 DC locations) deployed on large instances in the EC2 cloud platform. We deployed all the nodes within the same availability zone in EC2 and emulate the appropriate inter-DC delay using Dummynet [21].

8 Experimental Validation

In this section, we show the benefits of our framework and present the results from our experiments using Cassandra deployed on a test-bed.

8.1 Implementation

Off-the-shelf, Cassandra employs a random partitioner that implements consistent hashing to distribute the load across multiple storage nodes in the cluster. In the random partitioning scheme, the output range of a hash function is treated as a fixed circular space or ring. Each data item is assigned to a node by hashing its key to yield its position on the ring and the node assumes responsibility for the region in the ring between itself and its predecessor node on the ring, with immediately adjacent nodes in the ring hosting replicas of the data item. Cassandra allows applications to express replication policies at the granularity of keyspaces (partitions of data). We modify the application code to treat a data item or a group of data items as a separate keyspace and configure replicas for each keyspace. This enables us to choose different sets of replicas from the cluster for different data items. Note that the keyspace creation is a one-time process and does not affect the application performance.

8.2 Model accuracy and benefits

In this section, we compare the performance of our *BA* scheme when the strategy recommended by it was used in Cassandra, with predicted results from the model. We also compare the results with the performance of the default random partitioner. We use Twissandra, a twitter-like application for this experiment. We choose a user from the *Twitter* trace who has many friends spread across different geographical regions. The user is mapped to a DC in LA and her friends are mapped to 19 DCs including 12 in the US, 4 in Europe, 1 in Asia, 1 in Australia and 1 in South America. A majority of writes to the user's timeline arrive from LA and Seattle as her friends from other locations do not tweet actively. About 79% of reads come from LA and the remaining reads come in significant fractions from each of the other locations. When optimized for the $90\%ile$ of latency, the *BA* scheme chooses replicas in Dallas TX, Hayward CA and St.Louis MO with read and write quorums of size 2. Given that 90% of accesses come from the US West and East coasts, these choices are reasonable.

Figure 6, shows CDFs of the observed and predicted latencies of both reads and writes for the *BA* scheme. We see that the observed and predicted latencies (both reads and writes) for the *BA* scheme are very close to each other. The error in the $90\%ile$ observed latency is only 3.7% of 54ms (the model predicted latency) and it can be attributed to inherent variability in EC2.

Figure 6 also shows a CDF of the latency observed with the random partitioner of Cassandra. We see that the *BA* model performs better than the random partitioner. This is because the performance of random placement is intricately

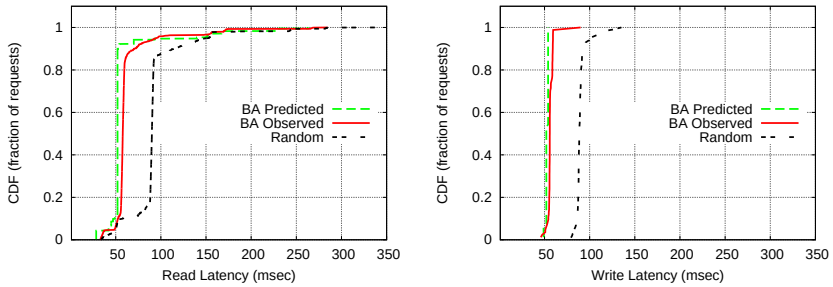


Figure 6: **EC2 Study: CDF of the observed read and write latency for all requests to the user timeline, comparing a random placement with the placement from the *BA* model. Each figure also shows the model predicted latency for reference.**

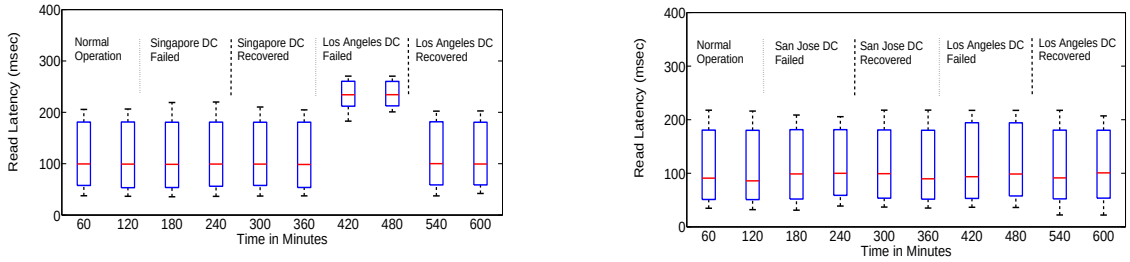


Figure 7: **EC2 Study: Boxplot showing the distribution of read latency per hour with the *Basic Availability* and the *N-1 Contingency* models.**

tied to the ring structure, whereas the *BA* scheme identifies an optimal placement. Overall these results validate our model and demonstrate its importance.

8.3 Model performance under failures

In this section, we conduct experiments to compare and study the performance of the *BA* and *N-1C* schemes under the failure of a DC. We use the English document key from the *Wikipedia* trace for this experiment. The accesses are spread across different geographic regions including about 50% from North America, 23% from Europe, 5% from each of Asia (mainly mapped to Singapore DC) and Australia as shown in figure 2.

The *BA* scheme places replicas in Los Angeles CA, Palo Alto CA and Singapore with read and write quorums of sizes 2. Since nodes in the US West are reasonably equidistant from Asia, Australia, Europe and US East, the *BA* scheme places 2 replicas in the West coast. Placing the 3rd replica in Singapore also reduces the $90\%ile$ latency by a small margin of 8ms under normal operation by reducing the delay for accesses from Singapore. Figure 7(a) shows the performance of the *BA* scheme under failure of different DCs. The X-axis shows the time in minutes and the Y-axis shows the distribution of the request latency for every one hour period. From the figure, we see that the $90\%ile$ latency increases significantly from 205ms (under normal operation) to 272ms when LA failed (33% increase), though performance is only slightly affected when the Singapore DC fails.

In contrast, the *N-1C* scheme explicitly optimizes for latency under a failure and places the 3rd replica in San Jose CA instead of Singapore. Figure 7(b) shows the performance of the *N-1C* scheme under failures of different DCs. The figure shows that even though *N-1C* incurs a modest penalty over *BA* during normal operation (the $90\%ile$ observed latency with *N-1C* increases by 8ms), the $90\%ile$ latency remains largely unaffected under all failures. These results highlight the need to explicitly optimize for performance under failure and show the benefits of *N-1C* over the *BA* scheme.

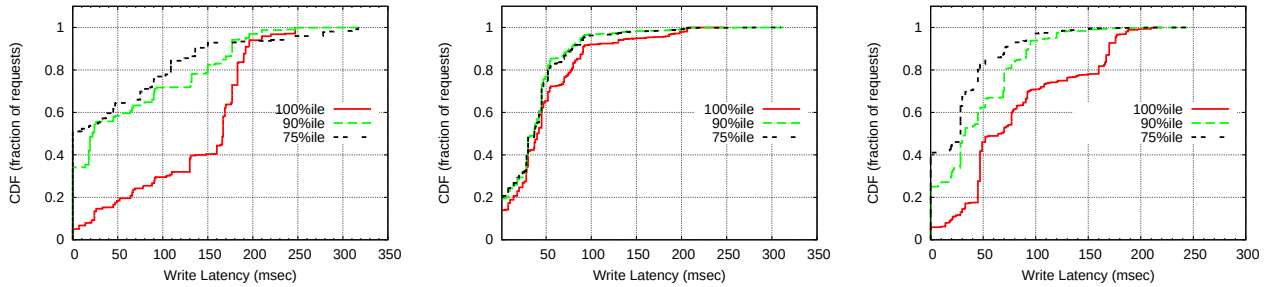
9 Simulation study

In the previous section, we validated the performance of our schemes using Cassandra deployed in an EC2 test-bed. We now explore the performance of our schemes on a larger scale by conducting extensive simulation study using the three real-world traces described earlier. We also study the sensitivity of our schemes to the various workload and model parameters. Unless otherwise mentioned all experiments shown in this section have been conducted using a trace of one month (Dec 2010) in *Twitter*, one month (Oct 2010) in *Gowalla* and one quarter (Q4 2011) in *Wikipedia*. Similarly, all experiments (except those in section 9.1) have been conducted by optimizing for the $90\%ile$ of requests for all keys.

9.1 What percentile to optimize for ?

In this section, we explore the sensitivity of the *LAT* scheme to the desired percentile of latency optimized (p^r and p^w in section 4.2.

Figure 8 shows the CDFs of the write latency observed by all requests (across all data items) for each of the three traces. Note that there is significant variation in popularity across data items. Each figure shows the observed write latency when placements for each data item are optimized for the 75th, 90th and 100th %iles of latency. We find that for



(a) *Wikipedia* (b) *Twitter* (c) *Gowalla*

Figure 8: **Observed write latency for all requests seen in each of the three traces *Wikipedia, Twitter* and *Gowalla* over a month. Each figure shows three curves corresponding to the percentile of latency for which the placements were optimized with *LAT* scheme.**

Wikipedia and *Gowalla*, optimizing for the $100\%ile$ hurts a majority of requests though the performance of the tail is marginally improved. This is because many keys have a large fraction of accesses originate from one region and a small fraction originates in a remote region. For e.g., for *Wikipedia*, over 90% of accesses to French articles originate from Europe but a small fraction originates from Canada. Optimizing for the $90\%ile$ results in replicas placed in Europe which benefits the majority of accesses, while optimizing for the $100\%ile$ places replicas in the US East Coast providing modest benefit for accesses from Canada, at the expense of requests from Europe. Similar trends were seen for read latency and the results are not shown.

Interestingly, for *Twitter*, there is no noticeable difference in performance when optimized for the three different percentiles. Further analysis showed that this is because 60% of data items have requests originating in exactly one DC. While requests for other data items arrive from multiple DCs, they come in comparable proportions which limits the opportunity for our model to improve the access latency by ignoring certain fractions of the accesses.

9.2 Can replication lower latencies ?

In this section, we consider the LAT scheme with symmetric read and write latency thresholds and evaluate to what extent replication can help lower latencies. Figure 9 shows the fraction of keys where the optimal latency was achieved using a replication factor greater than one for each of the traces. While we expect a majority of the keys to have just one replica, we found that a noticeable fraction (about 11% for *Twitter*, 7% for *Gowalla* and 3% for *Wikipedia*) benefited from a higher replication factor. We notice that *Twitter* and *Gowalla* has a larger number of cases where replication helps. We believe this is because in these traces reads and writes tend to come from different sets of DCs. On further investigation, we find that more than 20% of the keys were able to reduce their 90%ile latency by 18% or more through replication with some keys seeing reductions as high as 40%. These results are again consistent with observations in Section 4.3. Further, as we shall see in Section 9.5, the benefits due to replication can be higher with asymmetric read and write latency thresholds.

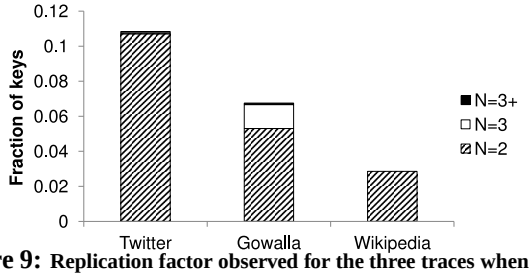


Figure 9: **Replication factor observed for the three traces when optimizing placement for the 90th percentile of requests with LAT**

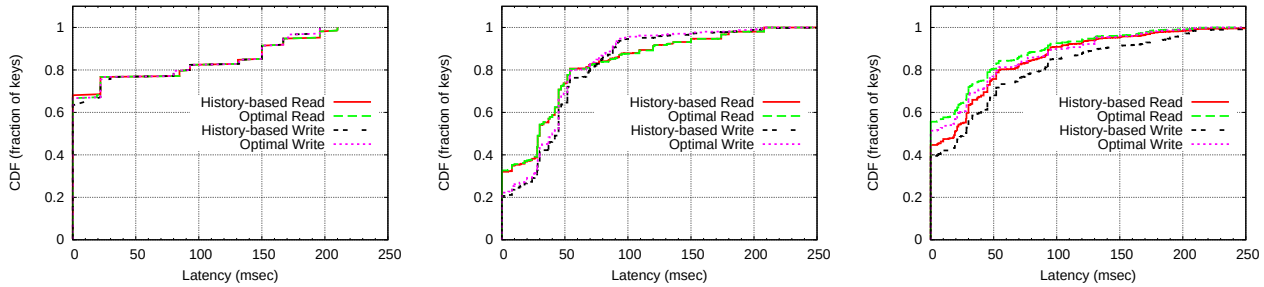
9.3 History-based vs Optimal

So far, we assume that the workloads and the access patterns for the applications are known. However, in practice, this may need to be obtained from historical data. In this section, we analyze this gap by comparing the performance of our schemes using historical and actual workloads for all three applications.

Figure 10(a) shows the CDF comparing the performance of *Wikipedia* during the first quarter of 2012 when using the history-based and the optimal replication configuration. The two curves labeled history-based correspond to the read and write latency observed when using the replica configuration predicted from the fourth quarter of 2011. The two curves labeled optimal correspond to the read and write latency observed when using the optimal replica configuration for the first quarter of 2012. Figures 10(b) and 10(c) show similar graphs for the *Twitter* and *Wikipedia* applications. From the figures, we find that the history-based configuration performs close to optimal for *Wikipedia* and *Twitter*, while showing some deviation from optimal performance for *Gowalla*. This is because users in *Gowalla* often move across geographical regions resulting in abrupt workload shifts. For such abrupt shifts, explicit hints from the user when she moves to a new location or automatically detecting change in the workload and rerunning the optimization are potential approaches for improving the performance.

9.4 Availability and failure analysis

In this section, we study the impact of availability requirements on the latency of the data-stores. We first analyze the additional latency incurred by the *BA* and *N-1C* schemes over the LAT scheme. We then explore the performance of



(a) *Wikipedia* (b) *Twitter* (c) *Gowalla*

Figure 10: Optimal performance for a given period with the performance using replica placements from the previous period for the three traces using LAT.

each model and compare the benefits of each scheme under normal and failure conditions.

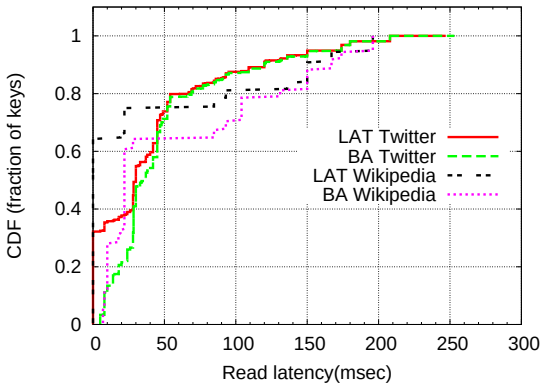


Figure 11: CDF comparing the read latency of the *Latency Only* and *Basic Availability* schemes.

Figure 11 compares the performance of the *BA* and LAT models for the *Twitter* and *Wikipedia* applications. We observe that some latency penalty is incurred by the *BA* scheme for both applications. Intuitively, this is expected since a higher replication factor would require the accesses to go to at least two DCs. While this penalty is marginal for *Twitter*, interestingly, we find it to be more pronounced at the initial percentiles for *Wikipedia*. On further investigation, we found that a large fraction of keys in *Wikipedia* had replicas configured in DC locations for which the closest DCs were more than 30msec away from them.

We now analyze the performance of the *BA* and *N-1C* schemes. Figure 12 shows the CDF of the read latency observed by both schemes for every key in *Twitter* and *Wikipedia* under normal and failure conditions. For each key, the figure plots the read latency under normal conditions (all replicas are alive) and when the most critical replica (replica whose failure would result in the worst latency) for the key fails.

From the figure, we see that the read latency observed by the *BA* scheme deteriorates quite drastically under failure for almost all keys in both the applications. For instance, more than 40% of the keys in *Twitter* observed an increase of 50msec or more under failure conditions. Similarly, in *Wikipedia*, more than 20% of the keys observed an increase of 100msec or more under failure. On the other hand, the read latency for *N-1C* observed only a marginal variation under failure. For instance, most keys in *Twitter* observed less than 30msec increase in latency on its replica failures. Surprisingly, we find that the *N-1C* scheme incurs an almost negligible penalty in its latency under normal conditions despite optimizing the replica configuration explicitly for the failure of a replica. On further investigation, we found that *BA* was often able to optimize latency with two of the chosen replicas and the third choice did not significantly impact performance. In contrast, the *N-1C* scheme more carefully selects the third replica ensuring good performance under failures. These results confirm and generalize our findings in Section 8.2.

Our results clearly show the benefit of explicitly optimizing the replication for failure conditions. The results highlight the benefits of our approach in ensuring good performance both under normal and under failure conditions.

9.5 Asymmetric read and write thresholds

Thus far, we assumed that reads and write latencies are equally desirable to optimize. However, in practice, some applications make prioritize read latencies, and others might prioritize writes. In this section, we explore solutions generated by our approach when the LAT model is modified to explicitly constrain the read and write thresholds.

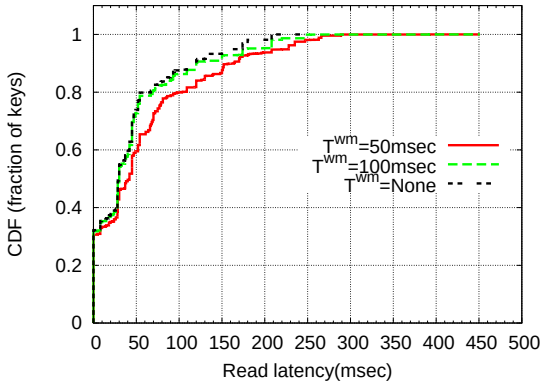


Figure 13: Read latency with different constraints on write thresholds for *Twitter*

Figure 13 shows the CDF of read latency for different con-

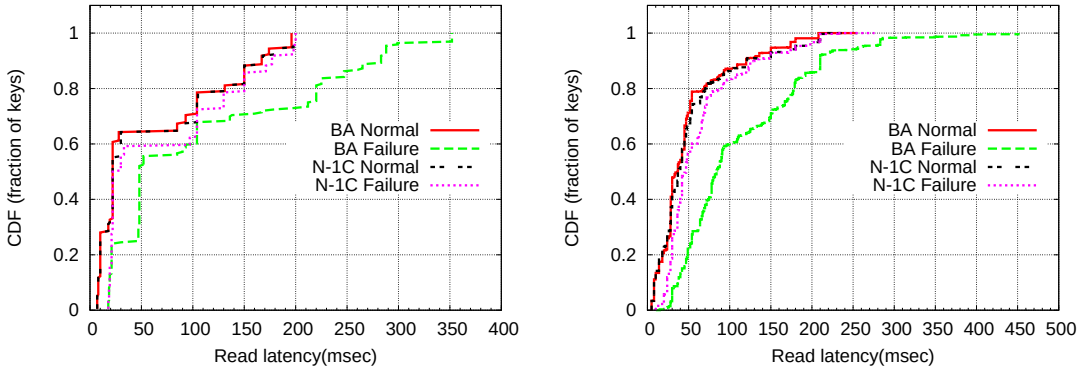


Figure 12: **CDF showing the read latency of the *Wikipedia* and *Twitter* traces under normal and failure conditions for the *BA* and *N-1C* schemes. Note that the curves for the two schemes under normal conditions overlap significantly.**

straints (T^{wm}) on the write latency. The $T^{wm} = None$ curve corresponds to our basic scheme without any constraints. From the figure, we see that a bound of 100msec on the write latency has no noticeable impact on the read latency, though the tail is more pronounced. Interestingly, we find that the bound of 50msec increases the read latency by less than 20msec for 60% of the keys.

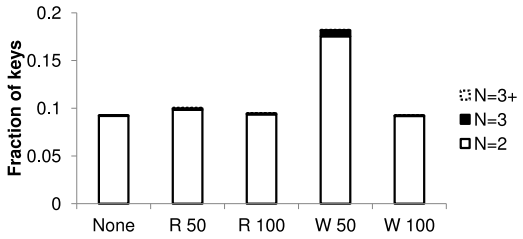


Figure 14: **Replication factor observed with *Twitter* with varying constraints on read and write latency.**

Figure 14 shows the histogram of the number of replicas suggested by our framework when various upper bounds are placed on the reads and writes. We see that $T^{wm} = 50$ resulted in configurations that had a significantly higher replication factor. This is expected because our LAT model tries to minimize the latency by moving the replica closer to the write locations (potentially impacting the read latency) in order to meet the constraint. The read quorum sizes were also found to be typically higher in these cases. Interestingly, a similar constraint on the read latency does not show this behavior. This is because, for most keys in our workload, reads tend to come from a single location and increasing the replication factor does not provide any additional benefit.

10 Related Work

Volley [14] seeks to place data across geo-distributed DCs considering inter-dependencies between data items. In contrast, we primarily focus on modeling replication protocols. While Volley supports user specified replication levels, our models automatically determine the number of replicas and parameters such as quorum sizes. Further, since we model requests arriving at storage coordinator nodes, our workloads already capture inherent inter-dependencies across items. For e.g., a user tweet that updates the timelines of a user and her friends would translate into our model as separate write events on each timeline object, and impact the placement of all timeline objects.

SPAR [43] presents a middle-ware for social networks which for each user ensures that the data of neighbors is co-located. SPAR achieves this property by having a master-slave arrangement for each data item, and ensures enough slave replicas are made. Geo-distributed cloud data stores support a broader range of applications, and seek to avoid issues related to data loss, temporary downtime, and higher write latencies that may be incurred by master-slave systems(Section 2).

Most geo-distributed cloud data stores in use today are adapted from weighted voting-based quorum protocols first proposed in [30], which allow replicas to have different weights in the quorum voting process. Although our models do not consider weights, because they are not used in geo-distributed cloud data stores, it is easy to extend our models to take weights into consideration.

A more flexible alternative to voting-based quorum protocols (termed coteries) was proposed in [29]. However, voting-based protocols are easier to implement [29], and more prevalent in operationally deployed cloud data stores.

Others have considered communication delays with quorum protocols [28, 45, 42]. In particular, [28, 45] consider problems that minimize the maximum node delays. However, these works are in the context of coteries, and do not directly apply to voting-based protocols. Further, these works do not consider consider optimizing latency percentiles, and latency under failures, both of which are key considerations for geo-distributed cloud data stores.

Several works have examined availability in quorum construction [18, 15, 35, 41, 22]. Most of these works do not consider the impact of failures on latency. Recent work [41] has considered how to dynamically adapt quorums to changes in network delays. Given that systems like Cassandra and Dynamo contact all replicas and not just the quorum, we focus on the orthogonal problem of replica selection so failure

of one DC does not impact latency. Several early works [18, 15] assume independent identically distributed (IID) failures, though non-IID failures are beginning to receive attention [35]. We focus on choosing replication strategies that are resilient and low-latency under failures of a single DC, since these are the more typical failure scenarios in our settings. Extensions of our model that consider simultaneous failure of DCs, for example those that connect to the network via the same point-of-presence, are straightforward.

11 Conclusions

In this paper, we have demonstrated the feasibility and importance of tailoring geo-distributed cloud data stores to meet the unique workloads of individual applications, so latency SLA requirements can be met during normal operations and on the failure of a DC. Our experimental evaluation with Cassandra shows that while latencies vary significantly (over a factor of two) across different replication configurations, our model recommended solutions perform within 4% of the predicted optimal. Further, our *N-1C* scheme reduces latencies by 55% on DC failures compared to *BA* while performing close to optimal under normal operation. These results highlight the importance of explicitly considering failures while choosing a replication configuration. Our simulations with longitudinal workloads of three widely deployed applications show that it is viable to choose replication configurations based on past workloads. Further, the results indicate the need for diverse replication configuration choices even across items within an application. Overall, these results highlight the need for a systematic framework like ours.

12 References

[1] Aws edge locations. `http://aws.amazon.com/about-aws/globalinfrastructure/`.

[2] Facebook's master slave data storage. `http://www.facebook.com/note.php?note_id=23844338919`.

[3] Geocoding in ArcGIS. `http://geocode.arcgis.com/arcgis/index.html`.

[4] Google app engine - transactions across datacenters. `http://www.google.com/events/io/2009/sessions/TransactionsAcrossDatacenters.html`.

[5] Google groups for App Engine Downtime Notification. `https://groups.google.com/forum/?fromgroups=#!forum/google-appengine-downtime-notify`.

[6] Latency - it costs you. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[7] More 9s please: Under the covers of the high replication datastore. `http://www.google.com/events/io/2011/sessions/more-9s-please-under-the-covers-of-the-high-replication-datastore.html`.

[8] Netflix rolls out Cassandra NoSQL data store for production in AWS. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`.

[9] Response Time Metric for SLAs. `http://testnscale.com/blog/performance/response-time-metric-for-slas`.

[10] Slow pages lose users. `http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html`.

[11] Stanford large network dataset collection. `http://snap.stanford.edu/data/loc-gowalla.html`.

[12] Twitter application uses key-value data store. `http://engineering.twitter.com/2010/07/cassandra-at-twitter-today.html`.

[13] Wikimedia statistics. `http://stats.wikimedia.org/`.

[14] S. Agarwal et al. Volley:automated data placement for geo-distributed cloud services. In *In Proc. NSDI*, 2010.

[15] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *In Proc. of The Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 26–35, June 1996.

[16] P. Bailis et al. Probabilistically bounded staleness for practical partial quorums. In *In Proc. VLDB*, 2012.

[17] J. Baker et al. Megastore:providing scalable, highly available storage for interactive services. In *In Proc. CIDR*, 2011.

[18] D. Barbara and H. Garcia-Molina. The reliability of voting mechanisms. *IEEE Transactions on Computers*, 36(10), October 1987.

[19] A. I. Barvinok. *A course in convexity*. American Mathematical Society, 2002.

[20] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Applications*, pages 1–29, 1982.

[21] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[22] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Optimizing vote and quorum assignments for reading and writing replicated data. In *In Proc. of The Fifth International Conference on Data Engineering*, 1989.

[23] E. Cho, S. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM, 2011.

[24] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, 2008.

[25] J. C. Corbett et al. Spanner:google's globally-distributed database. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex:a searchable distributed key-value store. In *In Proc. SIGCOMM*, 2012.

[27] D. Ford et al. Availability in globally distributed storage systems. In *In Proc. of OSDI*, 2010.

[28] A. W. Fu. Delay-optimal quorum consensus for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1), 1997.

[29] H. Garcia-molina and D. Barbara. How to assign votes in a distributed system. *Journal of the Association for Computing Machinery*, 32(4), 1985.

[30] D. K. Gifford. Weighted voting for replicated data. In *In Proc. SOSP*, 1979.

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

[32] P. Gill et al. Understanding network failures in data centers: Measurement, analysis, and implications. In *In Proc. of SIGCOMM*, 2011.

[33] F. Glover. Improved Linear Integer Programming Formulations of Nonlinear Integer Problems. *Management Science*, 22, Dec. 1975.

[34] D. Hastorun et al. Dynamo: amazons highly available key-value store. In *In Proc. SOSP*, 2007.

[35] F. Junqueira and K. Marzullo. Coterie availability in sites. In *In Proc. DISC*, 2005.

[36] U. G. Knight. *Power Systems in Emergencies: From Contingency Planning to Crisis Management*. John Wiley & Sons, LTD, 2001.

[37] A. Lakshman and P. Malik. Cassandra:a decentralized structured storage system. *Newsletter. ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.

[38] L. Lamport. Paxos Made Simple.

[39] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: unified and discriminative influence model for inferring home locations. In *KDD*, pages 1023–1031, 2012.

[40] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *In Proc. SOSP*, 2011.

[41] M. M.G., F. Oprea, and M. Reiter. When and how to change quorums on wide area networks. In *In Proc. SRDS*, 2009.

[42] F. Oprea and M. K. Reiter. Minimizing response time for quorum-system protocols over wide-area networks. In *In Proc. DSN*, 2007.

[43] J. M. Pujol et al. The little engine(s) that could: Scaling online social networks. In *In Proc. SIGCOMM*, 2010.

[44] A. Su et al. Drafting behind Akamai. *SIGCOMM 2006*.

[45] T. Tsuchiya, M. Yamaguchi, and T. Kikuno. Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes. *IEEE Transactions on Parallel and Distributed Systems*, 10(4), April 1999.

[46] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.