Samya: Geo-Distributed Data System for High Contention Data Aggregates

Sujaya Maiyya, Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi
University of California Santa Barbara
Santa Barbara, California
{*sujaya_maiyya, ishtiyaque, agrawal, amr*}*@cs.ucsb.edu*

***Abstract*—Geo-distributed databases are the state of the art to manage data in the cloud. But maintaining hot records in geo-distributed databases such as Google's Spanner can be expensive, as it synchronizes each update across a majority of replicas. Frequent synchronization poses an obstacle to achieve high throughput for contentious update-heavy workloads. While such synchronizations are inevitable for complex data types, simple data types such as aggregate data can benefit from reduced synchronizations. We propose an alternate data management system, *Samya*, to store and maintain aggregate data. It is presented as a system that stores cloud resource usage data. Samya dis-aggregates tokens of available resources and stores fractions of these tokens across geo-distributed sites. Dis-aggregation allows sites in Samya to serve client requests independently without the need to synchronize each update. Samya also incorporates a learning mechanism to predict the future resource demands at each site. If the predicted demand cannot be satisfied locally at a site, sites execute a synchronization protocol called *Avantan* to rebalance the available resource tokens in the system. Avantan is a novel fault-tolerant consensus protocol where sites agree on the global availability of resources that are then redistributed. After the redistribution, the sites continue to independently serve client requests. Our experiments, conducted on Google Cloud Platform, highlight that dis-aggregating data and reducing the number of synchronizations allows Samya to commit 16x to 18x more transactions than current state of the art cloud based geo-distributed databases.**

1. Introduction

Many small and mid-sized enterprises rely on large cloud providers, such as Amazon AWS, Google GCP, and Microsoft Azure, to provide the backend infrastructure. While the cloud's *pay-per-use* strategy along with the elasticity to spawn new resources on demand has many benefits, it comes with a cost: an unexpected traffic spike can drastically increase the consumed resources, leaving the customer with a hefty bill.

To avoid such surcharges, cloud customers can set limits on the amount of resources they consume through a variety of *resource tracking services* [8]. Clients can set limits on resources such as storage capacity, number of deployable VMs, and network bandwidth. Resource tracking services within a cloud provider actively maintain data on current resource usage; this data helps enforce the limits and bill the customer accurately for their usage. A resource can be consumed only if its current usage is below the preset limit of that resource – this translates to a *read-write* transaction at the resource tracking services.

Consider an example where a large cloud provider, *ultraCloud*, has a start-up *eCommerce.com* as a customer. The start-up comprises of many teams such as clothing, electronics, etc, as shown in Figure 1, and the teams consume resources as indicated in the leaf nodes. The resource limit is set by an admin of *eCommerce.com* and is applicable to all teams within the organization. This type of hierarchical structure is widely used by the cloud providers to allocate resources, track the usage, and accurately bill the customer [6], [1], [7].

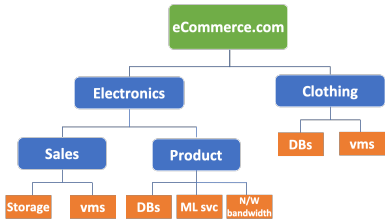

Figure 1: Hierarchical org structure of a cloud customer *eCommerce.com*.

The cloud provider, *ultraCloud*, tracks the number of resources *eCommerce.com* can consume. For example, each vm creation a read-write transaction in *ultraCloud*'s resource tracking service to check if the **overall** vms consumed exceeds *eCommerce.com*'s threshold. Only after this transaction succeeds can the actual physical resource be allocated. Any update to an intermediary or leaf unit (team) must percolate to the root node, *eCommerce.com*, as the cumulative resource usage by all the teams in the hierarchy is tracked at the root level. Typical update rates for a single node in the hierarchy may be in the hundreds of transactions per second, but the aggregate load on the root for a moderately sized enterprise hierarchy may easily be in thousands of transactions, causing the root node's data to become a ***hotspot***.

In a cloud setting, data – including a tracking service's data – are stored on *multiple* servers in a data center to ensure high availability and fault-tolerance. Examples of

geo-distributed databases are Google's Spanner [13] and Amazon Aurora [38].

Consider the design choices of a Spanner-like database: each data item is replicated across multiple sites, one of which acts as a leader. For each update, the leader replicates the change onto a set of replicas using consensus protocols such as Paxos [26]. While this is a good choice for high availability, it aggravates the hot-spot problem in two ways: (1) *Sequential execution*: for hot-spots, where many transactions access the same data, conflicting transactions are processed by the leader sequentially; and (2) *High Latency*: each update is propagated to geographically distant sites, incurring high latency. Spanner commits a transaction with a mean latency of 17ms and a tail latency of 75ms [13]; hence for a single data item, *Spanner can commit on average 58.8 transactions per second (tps) and a tail throughput of 13.3 tps*. For a customer such as eCommerce.com (Figure 1), perhaps 60 tps is enough for an individual node, but for the aggregate root node with hundreds of teams in the hierarchy, this throughput value becomes problematic.

Our observation is that while geo-distributed databases are a good choice for supporting complex forms of data, they are not ideal for simple aggregate data types where the operations are mostly limited to additions or subtractions, such as maintaining resource usage data. Spanner-like solutions provide high scalability but fail to provide the high throughput necessary for hot-spot data. Based on this observation, our research objective is to *design an alternate system that manages simple data types and provides high throughput for update heavy workloads in a cloud setting.*

This question has been addressed for traditional non-cloud databases in many works such as [31], [10], [25], [19]. Escrow transactions [31] introduced the notion of concurrent transactions updating different 'chunks' of the an aggregate data, albeit in a non-distributed database. Barbara et al. [10], Kumar et al. [25], and Golubchik et al. [19] introduced the idea of partitioning aggregate data onto multiple sites allowing each site to independently update its portion of the data value (e.g., multiple sites independently selling airline tickets). The problem is made non-trivial by introducing a constraint while updating the distributed data (e.g., not selling more airline tickets than the available seats). The solution proposed in this paper is motivated by these works, adapted for the radically different settings of large scale geo-distributed cloud infrastructures.

If partitions of available resources are to be stored on different sites, the next logical question to ask is: how to distribute the available resources among these sites? The advancements in machine learning and deep learning techniques as well as the abundance of cloud resource demand data collected by cloud providers can aid in answering this question. In fact if resource demand can be predicted and resources can be allocated to sites accordingly, most client requests can be served locally, without incurring expensive cross-datacenter communications.

In this paper, we propose an alternate design for geo-distributed data management systems to manage aggregate data. Specifically, we present ***Samya***[1] – a system that stores and tracks resource usage data across geo-distributed sites. Samya avoids the high latency and low throughput of Spanner-like databases by allowing a site to serve a client request locally, without the need for expensive synchronization.

Overview: To serve client requests locally while still maintaining the global resource limit, sites in Samya start with an initial allocation of available resources. We model the resource data as *tokens* (tokens of a specific resource are indistinguishable). A site can serve requests locally as long as it has locally available tokens; once it exhausts its local tokens or if it predicts an increase in resource demand that cannot be satisfied locally, the sites synchronize to *redistribute* any unused tokens in the system. We propose a novel protocol –*Avantan*[2] – to redistribute spare tokens.

Avantan is a fault-tolerant consensus protocol, in which, unlike Paxos, the value to agree upon is unknown at the start of the protocol. The sites communicate with each other to share their local token values and attempt to reach agreement on the shared values. If successful, the sites use the shared values to redistribute any spare tokens. Thus, sites in Samya are constantly rebalancing the tokens among themselves based on the demand predictions, to maximize the number of client requests served with minimal latency.

Along with providing low latency, the dis-aggregation strategy of Samya increases its availability compared to Spanner-like databases. For a specific resource, Spanner becomes unavailable if a majority of the sites that store the resource information fail, whereas Samya is available as long as at least one site is available.

Other Applications of Samya: Although Samya is motivated and presented as a service that stores and tracks resource usage, it can be used as a data managing system to maintain *any* aggregate data in the cloud. Examples of applications consisting of aggregate data are: rate limiting services to manage quotas and policies; inventory management such as online shopping, car rentals, etc.; airline ticket booking; advertisement campaigns tracking; billing services; etc,. For ease of exposition, in this paper we focus on one application: maintaining resource usage data.

The paper is structured as: Section 2 discusses existing works related to Samya, Section 3 discusses the system and data model employed in Samya. Section 4 explains transaction executions and introduces Avantan, Section 5 presents the experimental evaluation of Samya and Section 6 concludes the paper.

2. Related Work

The hotspot problem for aggregated data fields is an important practical problem studied extensively by the database community. Data partitioning is the most predominantly adopted solution for the hotspot problem, generally present in the two main forms: (i). *Key partitioning* where data

1. Samya is the Sanskrit word to equilibrium or equality.
2. Avantan in Sanskrit means allocation.

items are partitioned into different, non-overlapping sets based on their keys and the sets are stored across multiple sites; and (ii). *Value partitioning* where the same data item, irrespective of its key, is partitioned into different *values* and these values are stored across multiple sites. Since Samya is designed for a single high contention hotspot data, such as the root of an organization hierarchy, Samya adopts the value partitioning approach to store fractions of an aggregate value (i.e., available tokens) across different sites.

The idea of value partitioning has been studied extensively, starting with the seminal paper by O'Neil [31]. In [31], O'Neill introduced escrow transactions where different transactions operate on different fractions of the same data, thus allowing concurrency; this was proposed for a non-distributed database. In [25], Kumar and Stonebreaker extended transactions acquiring escrows to sites acquiring escrows. The sites serve transactions locally as long as they have non-zero escrow quantity. In [21], Harder extended the idea of escrows and introduced hierarchical escrows to reduce coordination to dynamically update escrows of multiple sites. In [24], Krishnakumar and Bernstein proposed Generalised Site Escrow to dynamically allocate parts of aggregate data (i.e., resources) to different sites using quorum locking and gossip messages.

The demarcation protocol [10] introduced by Barbara and Garica-Molina partitions an individual data value (which has a global constraint) and stores different partitions on separate machines; the protocol explains how to maintain constraints on the data when the data is distributed. In [9], Alonso and El Abbadi extend the demarcation protocol to store the value partitions across more than two sites and formalize the theory of partitioned data. In [19], Golulbchik and Thomasian introduce a token allocation system assuming that the incoming request pattern follows a Poisson distribution and tokens are allocated to different sites based on this distribution.

In essence, the above discussed works aim to partition a data item based on its value, store the partitions on multiple sites, and update them concurrently, while maintaining a global constraint. These protocols are proposed for radically different environments where typically the sites are not geo-distributed, the networks are assumed reliable, and the results presented are typically via simulations. Samya brings the basic idea – *dis-aggregate the aggregate data to increase concurrency* – from these works into the more modern context of cloud computing and geo-distributed data management systems.

A related approach that supports local operations without global synchronization is proposed by Shapiro et al. [35] in the context of Conflict-free Replicated Data Types (CRDTs). CRDTs supports conflict freedom by using eventually consistent replicas on different sites. Due to the eventual consistency guarantees and the semantics of the data types, replicas are updated locally and are synchronized with other replicas eventually. CRDTs, or rather systems that use CRDTs, differ from Samya in that the replicas of CRDTs do not dis-aggregate the value of a data item nor maintain a global and distributed constraint, which are important aspects of Samya. All the replicas of a data item in CRDT systems eventually become consistent with each other, without a notion of re-balancing the values maintained by each replica, as is performed in Samya. Another major difference between CRDT systems and Samya is CRDTs are typically commutative whereas data in Samya can be non-commutative.

Many recent works have proposed key partitioning as a way to scale and improve the throughput of database systems, such as Schism [15], Horticulture [33], Clay [34], E-Store [36], and Chiller [39]. All of these works differ from Samya in two major ways: (i). they partition the data records and optimally place hot records on different sites (except Chiller [39] which places hot records on the same site) to load balance and increase system performance; and (ii) they optimize for a set of hot records by opting for key partitioning, whereas Samya optimizes for a single hot record by choosing value partitioning.

With the rise of the cloud paradigm, many new database designs opt for geo-distribution to provide high availability [13], [16], [38], [2], [37]. The data in these systems are key partitioned and replicated across geo-distributed sites. Google Spanner [13], Amazon Aurora [38], and CockroachDB [37] all use replication protocols such as Paxos [26] or Raft [32] to consistently replicate each update to a quorum (typically majority). While Amazon's Dynamo [16] chooses eventual consistency and is hence less stringent in replicating each update, it may suffer from inconsistent data. Recently, there has also been increasing interest in efficiently executing transactions on data that is *both* partitioned and replicated in a cloud geo-distributed environment, in works such as MDCC [23], TAPIR [40], Replicated Commit [27], G-PAC [28], and Janus [29].

In general, these approaches differ from Samya in that they employ key partitioning and aim to efficiently execute distributed transactions across these partitions, which are often also replicated. First, due to replicating each update on to a quorum of geo-distributed sites, all of the above systems are prone to hot-spot problems for update heavy and contentious workloads. Second, the design mantra common across these works is to build a general data management system that can store varied and complex forms of data. While the general approach has many benefits, it fails to take advantage of application specific data forms (such as aggregate data fields) to optimize performance. This causes applications such as cloud resource tracking services to inevitably design their own data managing systems. Samya is designed to take advantage of aggregate data to provide high performance for hot-spots without compromising availability.

3. Samya Architecture

3.1. System Model

Samya is a niche distributed data management system that stores and maintains aggregate data specifically related

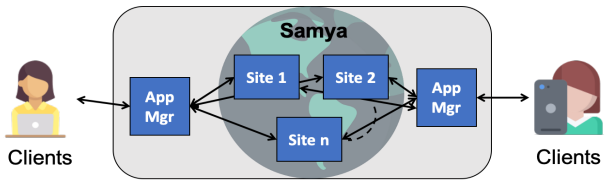

Figure 2: Clients interact with the data stored across sites through application managers (app mgr).

to resource usage information; the data is stored across multiple geo-distributed sites. Figure 2 represents the system model and the client interactions with the system. Samya consists of *sites* and *application managers*.

Sites: To enhance the performance and for high availability, the aggregate data is dis-aggregated into different partitions and the partitions are stored across multiple sites, typically geo-distributed. Sites in Samya act as *data shards* that store fractions of available resources and partial usage information of a resource; and for simplicity, we assume that all sites store information about all resources. Changing this design choice to allow only some sites to store information of specific resources is fairly straightforward, and furthermore, a run-time library can provide lookup and directory services to identify the sites that store a specific resource.

Application Managers: These are stateless processes that relay the messages between a client and the sites. App managers mask the network topology and individual site availability from external clients. Since the sites storing the data and the clients accessing the data are geo-distributed, multiple geo-distributed app manager processes exist to reduce the communication latencies between clients and sites. Being stateless, app manager processes can easily scale on demand depending on the request load.

Samya assumes an underlying asynchronous communication network where messages can be delayed, dropped, or reordered. The sites and the application managers can fail by crashing but do not exhibit malicious behavior. Unless they crash, the sites and application managers execute the designated protocol correctly. Samya further assumes that a site, which stores the data, *does not crash indefinitely*; when a crashed site recovers, it reconstructs its previous state before the crash. If an application manager crashes, since app managers are stateless, a new process can be spawned easily and plugged into the system.

3.2. Data Model

Abstractly, we term each resource stored in Samya as an *entity*. The clients (i.e., cloud customers) acquire or release these entities and Samya tracks client actions to maintain up-to-date resource usage and resource availability information for each entity. A high privilege-user (e.g., admin of an enterprise) within a client configures a preset maximum $\mathcal{M}_e$ – the maximum limit of available tokens for entity e – and other clients (e.g., smaller organizational units in the enterprise or end users of the enterprise) can acquire or release specific quantities of the entity.

Samya maintains the following system level constraint: at no point does the system allow the clients to collectively acquire more than $\mathcal{M}_e$ tokens for an entity e.

$$0 \leq total\_acquired\_tokens \leq \mathcal{M}_e \quad (1)$$

TABLE 1: State variables of an entity e maintained by each site in the system.

item	*description*
id	*UUID* to identify type of resource
$TokensLeft_S$	Num. of tokens left at site S
$TokensWanted_S$	Num. of tokens site S wants

The state of an entity e, as maintained by each site S in the system, refers to specific details as presented in Table 1: id is a unique identity to identify the type of entity (or resource) e; $TokensLeft$ indicates the number of tokens of entity e available at site S; $TokensWanted$ indicates the number of tokens of entity e that site S needs during a redistribution.

Transactions: Clients perform 2 types of transactions:

- *acquireTokens(e, n)*: A client asks for n tokens of entity e, where n is a positive integer.
- *releaseTokens(e, m)*: A client returns m tokens of entity e back to the system, where m is a positive integer. These tokens can later be acquired by other clients. An individual client never returns more tokens than what it has acquired.

4. Samya

In this section we discuss how Samya efficiently manages and tracks resource usage. Samya is a highly available distributed data management system proposed as an alternative to manage resource data in geo-distributed databases. If a client consumes any resource such as creating additional VMs, then in traditional geo-replicated databases, all the replicas are updated to reflect the resource usage. Samya, on the other hand, chooses a single site to update the resource usage data, thus avoiding the cross data-center communications for each update. To cope with varying resource demands at different sites, Samya relies on learning based predictions and dynamic reallocation of resources.

4.1. Overview

In this section, we provide an overview of Samya's request serving approach. A site receives either an *acquireTokens(e, n)* or *releaseTokens(e, m)* request from a client, where e identifies the entity, and n, m are the number of tokens to be acquired or released. The main goal of each site in Samya is to serve as many requests locally as possible while maintaining the global constraint that the overall acquired tokens of an entity e stored across all sites never exceeds the limit $\mathcal{M}_e$. Since a *releaseTokens* request returns tokens, it never violates the global constraint and hence, can always be served locally at a site.

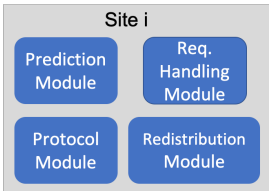

Figure 3: Components of each site in Samya.

Meanwhile, a site may receive an acquire request with a value greater than the number of tokens available locally at that site. A site could choose to reject these requests but Samya takes an alternate approach. If a site S cannot serve an acquire request locally, it triggers a *redistribution* by communicating with other sites. The sites share the state of their tokens for entity e and redistribute any spare tokens, after which site S may acquire enough tokens to serve pending or future client requests. We term this as *reactive redistribution*: a redistribution triggered in response to a client request that could not be satisfied locally

While reactive redistributions avoid rejecting client requests merely because tokens are exhausted locally, the requests that cause redistributions incur large delays. The cloud computing literature [20], [14], [30], [17], [22], [12] has shown that resource demand typically can be predicted. We take advantage of predictable workloads to trigger *proactive redistributions* – redistributions where a site predicts if the demand is increasing and triggers a redistribution to satisfy the predicted load. This approach further minimizes the latency to serve client requests.

4.1.1. Components of a Site in Samya. Each site in Samya consists of 4 components as shown in Figure 3.

- **Request Handling Module**: This module communicates with app managers and serves client requests locally. This module also triggers redistributions.
- **Prediction Module**: This is a learning module generated by training on existing resource demand data such that it predicts the future resource demand in terms of number of tokens.
- **Redistribution Module**: If the Request Handling module triggers a redistribution, this module calls the Protocol module to check if other sites have any spare tokens; based on the responses from other sites, this module re-allocates the spare tokens.
- **Protocol Module**: This module executes a multi-round fault-tolerant protocol that collects token information from other sites for redistribution.

Each of these modules is pluggable and can be easily replaced with an upgraded version, if and when needed.

4.1.2. Life-cycle of a client request. A step-wise overview of how sites in Samya serve client requests is presented in Figure 4:

1). Client request: A client generates and sends either an *acquireTokens(e, n)* or *releaseTokens(e, m)* request, where e is the entity id, and n, m are number of tokens. This request

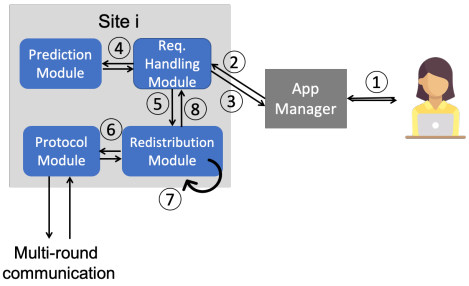

Figure 4: Life-cycle of a client request.

reaches the closest app manager to the client (this can be achieved using a load balancer).

2). App Manager: Typically, the app manager relays the client request to the closest site. But if the closest site has failed or is overloaded, an app manager may relay the client request to another site. As a result, a single client's *acquireTokens* request may be sent to a site S_k whereas a *releaseTokens* request may be sent to a different site S_j. This is acceptable because sites in Samya only store the resource usage data; Samya is not responsible for the actual physical resource allocation, which is the function of higher level applications.

3). Site serving request: A site S that receives a client request attempts to serve the request locally; if successful, it updates its local token state based on the type of request:

$$TokensLeft_S = TokensLeft_S - n \tag{2}$$

if the client acquired n tokens; or

$$TokensLeft_S = TokensLeft_S + m \tag{3}$$

if the client released m tokens. The site then responds to the client, which is relayed via an app manager.

4). Demand prediction: After serving a client request, the site checks if it is close to exhausting its tokens for that entity, i.e, *TokensLeft* is below a pre-configured threshold. If so, the site predicts the demand for the near future (e.g., next 5 minutes).

5). Trigger redistribution: If the predicted value indicates a decrease in demand, the site simply continues to serve more requests. Whereas if the predicted value indicates an increase in demand that cannot be satisfied locally, the site triggers a redistribution.

6). Execute protocol: If the site triggers a redistribution, it communicates with other sites to collectively execute a fault-tolerant protocol to share with each other the state of entity *e*, which includes the information shown in Table 1.

7). Reallocate tokens: Based on the shared information, each site independently reallocates the overall spare tokens using a deterministic reallocation procedure.

8). Update tokens state: Depending on the outcome of the reallocation, the site that triggered the redistribution may acquire more tokens, upon which it updates its state of entity e and serves any pending or future requests.

Note that the above steps describe a *proactive* redistribution. Samya also supports a *reactive* redistribution triggered

when a site receives a client request that cannot be served locally (due to insufficient locally available tokens). Cloud workloads typically consist of spikes and a reactive approach caters to such spiky workloads.

In the following sections, we elaborate on when a redistribution is triggered, how the redistribution protocol is executed, and once the protocol terminates, how the spare tokens are reallocated. Table 2 defines the variables that will be used in the following sections.

TABLE 2: Variables used in the t^{th} redistribution of an entity e at site i.

Symbol	Meaning
$\mathcal{M}$	Maximum Limit (of entity e)
N	Number of sites
TU_t	Tokens Used at t^{th} redistribution
TL_t	Tokens Left at t^{th} redistribution
TW_t	Tokens Wanted at t^{th} redistribution
$\mathcal{S}_t$	Total spare tokens in t^{th} redistribution
$\mathcal{R}_t$	Set of sites participating in t^{th} redistribution
$\mathcal{L}_t$	List of state variables of the sites in $\mathcal{R}_t$

4.2. Triggering Redistribution

Before delving into the details of triggering a redistribution, we discuss predictability of cloud workloads. The cloud computing literature consists of many works that highlight the predictability of resource demands in the cloud, e.g., [20], [14], [30], [17], [22], [12]; they also discuss various techniques to predict resource demand. The common underlying idea is to collect a large amount of actual demand data, analyze this data to uncover any periodicity or patterns, and develop mathematical models that can learn from the past data to predict future demands.

Samya adopts a similar approach where application-specific historical resource demand data is collected to train a learning model. Once this model is trained and tuned to predict future demands, it is used as the Prediction Module (Figure 3). The Prediction Module is a pluggable module wherein the application developers using Samya are free to choose the best prediction technique suitable for their workload. This module can be replaced even after deployment, if a better learning approach is found or if the application workload changes. We discuss the prediction methods used in evaluating Samya in Section 5.

An *epoch* is defined as the look-ahead time duration used during prediction. This dictates how far ahead in the future to predict resource demand (e.g., 5 or 10 minutes) depending on the workload pattern. Samya supports two ways of triggering a redistribution.

- *Proactive redistribution*: After a site serves an *acquireTokens(e,n)* request, in a background thread, it checks if it is close to exhausting its local tokens for that entity. If so, it uses the Prediction module to predict resource demand for the next epoch. If demand is decreasing, the site continues to serve client requests. Whereas, if demand is to increase in the next epoch such that it cannot cater to the increasing demand locally, the site triggers a redistribution by updating its state of entity e:

$$TokensWanted = PredictedValue - TokensLeft \quad (4)$$

- *Reactive redistribution*: Since prediction techniques are rarely 100% accurate, Samya allows for *reactive* redistribution wherein a site receives an *acquireTokens(e,m)* request asking for m tokens and $m > TokensLeft$ at the site. In order to satisfy this request, the site triggers redistribution by updating its state of entity e:

$$TokensWanted = m \quad (5)$$

4.3. Executing Redistribution Protocol

Once a site decides to trigger redistribution, it executes *Avantan*: a novel fault-tolerant consensus protocol designed specifically for redistributing available resources. In this section we present two different versions of *Avantan* differing primarily in their failure assumptions and failure recoveries. Sites execute multiple instances of Avantan either sequentially or concurrently; a single execution instance is presented in this section. For each instance of redistribution, the Avantan protocol aims to reach agreement on the list of sites participating in that instance. The protocol is designed to tolerate arbitrary crashes, message losses, and network partitions, while making the best effort in providing liveness.

The two versions of Avantan are:

- **Avantan[$\frac{n+1}{2}$]**: Requires a majority ($\frac{N}{2}+1$) of sites to be alive and communicating during protocol execution. This version of Avantan is a better choice when individual network links are highly unreliable (prone to message drops) and servers crash frequently but network partitions are infrequent. In this version all sites execute one redistribution after another.
- **Avantan[$*$]**: No requirements on majority of sites being alive to execute the protocol; it tolerates network partitions of arbitrary sizes and allows different partitions to execute redistribution concurrently. But this version is sensitive to message losses during the execution of the protocol.

The two versions also differ in their failure recovery mechanism, which will be discussed later. In developing the protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [28] and the protocol is motivated by the Paxos Atomic Commit (PAC) protocol proposed in [28]. Avantan abstractly consists of the following phases: the first phase executes *Leader Election* as well as *Value Construction*, the second phase makes the value *Fault-Tolerant*, and finally, the third asynchronous phase distributes the *Decision*.

We explain the two versions of Avantan with respect to redistributing the tokens of a single entity e; the protocol can be easily extended to include multiple entities. During

Algorithm 1 Avantan[$\frac{n+1}{2}$] redistribution protocol.

Let $state_{t.i}$ be the state of entity e at site with id i during t^{th} execution of Avantan.

1: **procedure** ELECTION-GETVALUE()

2: BallotNum ← (BallotNum.num+1, selfId)
3: InitVal ← $currState$ /* With an updated $TokensWanted$ */
4: Send *Election-GetValue(BallotNum)* to all

5: **procedure** ELECTIONOK-VALUE()

6: upon receiving *Election-GetValue(bal)* from S
7: **if** bal > BallotNum **then**
8: BallotNum ← bal
9: predictedVal ← PredictForNextEpoch()
10: **if** predictedVal > currState.TokensLeft **then**
11: currState.TokensWanted ← predictedVal - currState.TokensLeft
12: InitVal ← $currState$
13: Send *ElectionOk-Value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision)* to S

14: **procedure** ACCEPT-VALUE()

15: **if** received *ElectionOk-Value(bal, initV, acceptV, acceptN, dec)* from majority **then**
16: **if** at least ONE response with *dec*=True **then**
17: AcceptVal ← *acceptV* of that response
18: Decision ← True
19: **else if** at least one response with *acceptV*$\neq \perp$ /* *dec* is True for none. */ **then**
20: AcceptVal ← *acceptV* of that response
21: **else**
22: AcceptVal ← (InitVal || all received $initVs$)
23: **end if**
24: AcceptNum ← BallotNum
25: Send *Accept-Value(BallotNum, AcceptVal, Decision)* to all

26: **procedure** ACCEPT-OK()

27: upon receiving *Accept-Value(bal, acceptV, acceptN, dec)* from S
28: **if** bal ≥ BallotNum **then**
29: AcceptVal ← acceptV
30: AcceptNum ← bal
31: Decision ← dec
32: Send *Accept-ok(BallotNum)* to S

33: **procedure** DECISION()

34: **if** received *Accept-ok(bal)* from majority **then**
35: Decision ← True
36: Send *Decision(BallotNum, Decision)* to all

TABLE 3: Variables maintained by a site with id s during each execution of redistribution protocol for entity e.

BallotNum	initially $< 0, s >$
InitVal	state of entity e (Table 1)
AcceptVal	initially $\perp$ (Null)
AcceptNum	initially $< 0, s >$
Decision	initially False

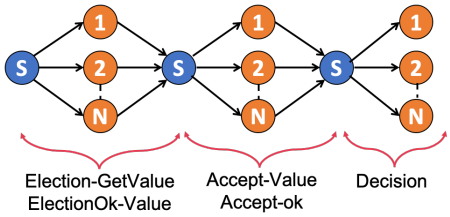

Figure 5: Phases of Avantan[$\frac{n+1}{2}$] protocol.

protocol execution, sites maintain the variables defined in Table 3, which mainly correspond to the standard variables used in Paxos. $BallotNum$ is a tuple of the form $<num, id>$ where *num* is a local, monotonically increasing integer and *id* is site id. Ballot number ensures the total ordering of different redistributions. *InitVal* is the current state value of entity e (as defined in Table 1) at site s when the redistribution starts. *AcceptVal* represents the list of state values of the sites participating in the redistribution and *AcceptNum* is the ballot number at which a site updates its *AcceptVal*. Finally, $Decision$ indicates if the sites reached agreement on *AcceptVal* at ballot $BallotNum$.

The redistribution protocol is initiated by a site S either for proactive or reactive reasons. The different phases of the protocol are shown in Figure 5. Abstractly, site S attempts to become the leader and collects state values from: at least a majority of the sites in the case of Avantan[$\frac{n+1}{2}$]; and any number of sites in Avantan[$*$]. We denote the set of sites participating in the t^{th} instance of redistribution as $\mathcal{R}_t$[3]. Site S then ensures that the list of state values – as denoted by $\mathcal{L}_t$ – is fault-tolerantly stored across: a majority of sites in Avantan[$\frac{n+1}{2}$]; and the same set of sites that responded with their state values in Avantan[$*$]. S then finalizes the value $\mathcal{L}_t$ and all participating sites reallocate the tokens. Note that once a site starts participating in the redistribution protocol, it queues all the *acquireTokens* and *releaseTokens* requests from clients until the protocol terminates.

4.3.1. Avantan[$\frac{n+1}{2}$]. The protocol consists of 3 rounds (5 phases) as described in Algorithm 1 and shown in Figure 5:

- ***Election-GetValue***: In the first phase site S attempts to become the leader as well as collect the state values from other sites. Site S increments its ballot number (line 2) and sets its $InitVal$ to the current local state of entity e, i.e., all the fields of *TokensUsed, TokensLeft, and TokensWanted*. Site S then sends Election-GetValue(BallotNum) message to all sites.

3. These variables are defined in Table 2.

- ***ElectionOk-Value***: As shown in lines 6-13, upon receiving the *Election-GetValue* message, a site C (termed as cohort to distinguish from the leader) checks if the received ballot number is greater than its current ballot number. If yes, it updates its ballot number, and it runs the Prediction Module to predict its demand for the next epoch (line 9). If the predicted value is greater than the current number of tokens left, then site C's demand is increasing such that C cannot satisfy the increasing demand. Hence, it sets its $TokensWanted$ field (line 11) to the difference between the predicted value and its locally available tokens. C then sets its $InitVal$ to the updated state and sends ElectionOk-value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision) to leader S. The *AcceptVal, AcceptNum*, and *Decision* variables are used in failure recovery; in a failure-free execution, these variables are set to the initial values as defined in Table 3.
- ***Accept-Value***: As shown in lines 15-25, the leader site S waits until it receives ElectionOk-Value messages from at least a majority of the sites (including itself). In a failure-free execution (failure recovery explained later), S sets $AcceptVal$ to the concatenated *InitVal*s received in the ElectionOk-Value responses (line 22), and sets $AcceptNum$ to its current ballot number. S then sends Accept-Value(BallotNum, AcceptVal, Decision) to all sites.
- ***Accept-ok***: Upon receiving the Accept-Value message from the leader, indicated in lines 27-32, a cohort C checks whether the received ballot number is at least as high as its current ballot number. If yes, it updates the $AcceptVal$, $AcceptNum$, and $Decision$ variables and sends an Accept-ok(BallotNum) message to the leader.
- ***Decision***: Finally, the leader waits for Accept-ok messages from at least a majority of sites (including self), then sets its $Decision$ variable to True, and sends the Decision(BallotNum, Decision) message to all. The protocol terminates for the leader when it sends the $Decision$ message; whereas the protocol terminates for a cohort once it receives the Decision message. The sites then reallocate the tokens using the information in $AcceptVal$ and respond to any pending client requests that were queued. The sites also reset the variables defined in Table 3 (except $BallotNum$) once the protocol successfully terminates.

Failure Recovery: If the leader crashes or is network partitioned from the rest of the sites, the sites must recover in order to continue serving the clients. The failure recovery execution follows the same steps as a failure-free execution: if a site S' times-out waiting for the leader's message, S' attempts to become the new leader and terminate the protocol by sending Election-GetValue message to all the sites. As shown in Procedure Accept-Value (lines 16-28), in the received ElectionOk-Value messages, if S' receives at least one message with $Decision$ as True, this implies that the previous leader had terminated the protocol and had sent at least one Decision message before failing; so S' chooses the $AcceptVal$ received in this message (lines 18-20).

If none of the received messaged has $Decision$ as True but at least one message has a non-empty $AcceptVal$, this implies that the previous leader had received all the $InitVal$s and constructed the $AcceptVal$ and had sent Accept-Value to at least one site before failing; hence the new leader S' chooses this value as $AcceptVal$ (lines 21-22). If multiple sites respond with differing $AcceptVal$s, the new leader chooses the $AcceptVal$ corresponding to the highest $AcceptNum$. Any other case implies the previous leader had either failed to construct $AcceptVal$ or to store it on a majority before failing, and hence, S' is free to construct $AcceptVal$ based on the received $InitVal$s (line 24) (this is also the failure-free behavior). The next steps of fault-tolerantly storing the chosen value and sending the decision are the same as in failure-free executions.

Fault Tolerance: As stated in the FLP impossibility result [18], no consensus protocol can guarantee termination even with a single site failure. Following the impossibility result, Avantan[$\frac{n+1}{2}$] can block if a majority of the sites fail or are unreachable, similar to Paxos.

In spite of the blocking behaviour of Avantan[$\frac{n+1}{2}$], the availability of Samya is higher than that of a system that executes Paxos for each transaction (e.g., Spanner). This is because the Avantan[$\frac{n+1}{2}$] protocol does not block if a majority of the sites have failed in the *first* phase of the protocol. To provide liveness, we use timeouts: if a site that wants to be a leader sends out Election-GetValue message but does not receive enough ElectionOk-Value messages within the timeout period, the site terminates the redistribution and continues to serve any client requests that can be served locally. This is acceptable since the leader failed to construct any value before aborting the redistribution.

Whereas, if the leader successfully constructed a value (after receives enough ElectionOk-Value messages) and sent Accept-Value messages to all sites but it failed to make the value fault-tolerant, i.e., it did not not receive enough Accept-Ok messages, then that site and the other live sites are blocked until a majority recover.

Theorem 1*: No two distinct values are both chosen for a given instance of Avantan[$\frac{n+1}{2}$].*

The recovery mechanism of Avantan guarantees *safety* of the value – if a majority of the sites accepted the value by sending Accept-ok message, then no site will agree on a different value for that instance of redistribution. This guarantee is ensured because there exists at least one overlapping site in the two sets of majority used in the first and second phase of the protocol and any new leader learns of a value that was chosen by the previous leader through the overlapping site. This guarantee holds as long as majority of the sites are alive and reachable. □

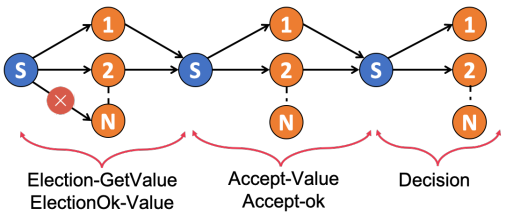

Figure 6: Phases of Avantan[$*$] protocol.

4.3.2. Avantan[$*$]. Avantan[$\frac{n+1}{2}$], similar to Paxos [26] and other other consensus algorithms, is restrictive as it requires communication among a majority of sites for redistribution to succeed. If a majority of the sites are down or if the network partitions such that no partition has a majority, then the sites cannot redistribute tokens and may end up rejecting many client requests. However, the token requirements of a site S, as represented in the $TokensWanted$ field of its state, might be satisfied by fewer than a majority of sites.

The logic of redistributing tokens among a set of sites does not impose any requirements on the minimum number of sites. Hence, we propose an alternative consensus protocol that allows *any subset of sites* to participate and ensures that all participating sites agree on the same value. We modify Avantan[$\frac{n+1}{2}$] to accommodate these new requirements.

The failure free execution of Avantan[$*$] is the same as the one presented in Algorithm 1 but with 3 major changes:

(i). The leader S that triggers the redistribution sends Election-GetValue messages to all sites. But instead of waiting for responses from a majority of sites, it waits until it receives ElectionOk-Value messages (with $TokensLeft$ field set) such that S's token requirements can be satisfied; if after a predefined amount of time, if S does not receive enough responses, it aborts the redistribution and notifies other sites and the client (for reactive redistributions). All the sites whose $InitVal$s were collected form the set $\mathcal{R}_t$ – the set of sites participating in t^{th} redistribution; in all subsequent rounds, S communicates only with the sites in the $\mathcal{R}_t$, while notifying the other sites to discard this redistribution.

(ii). If a cohort site responds with ElectionOk-Value message to one leader, it rejects all other Election-GetValue messages from concurrent leaders (even if they have higher ballot) until the former instance of Avantan[$*$] is complete. This ensures that a site participates in one instance of redistribution after another.

(iii). Rather than wait for any majority of sites to respond with Accept-ok messages (as shown in line 37), the leader S waits to receive Accept-ok from *ALL* the sites in $\mathcal{R}_t$ before it sends out the decision.

Different sets of sites can execute parallel redistributions but an individual site participates in one redistribution at a time.

Failure Recovery: Since Avantan[$*$] does not require a majority quorum to proceed, its failure recovery differs from that of Avantan[$\frac{n+1}{2}$]. In Figure 6, the leader S or other participants can fail at any point during the execution of the protocol. Sites such as site N, that did not even receive the Election-GetValue message are free to participate in other redistributions. If the leader fails (crash or network partition), sites such as 1 and 2 that participated in the redistribution, must be able to recover.

A cohort site C that participated in the redistribution detects leader failure using time-outs. Upon timeout, C checks the progress of the protocol execution using the variables defined in Table 3. If site C's $Decision$ variable is set to True, this implies the protocol had terminated and so C reallocates the tokens. If the $Decision$ is not true, C decides its next action based on the value of $AcceptVal$:

i). If $AcceptVal = \perp$: This implies that C did not receive AcceptVal from the leader S before S failed; thus, C is free to abort this redistribution because the previous leader could not have proceeded to the *Decision* phase without the Accept-ok from C.

ii). If $AcceptVal \neq \perp$: This implies that the leader had chosen a value but may not have decided on it, as the leader may have failed to receive enough Accept-oks before failing. In this case, C contacts all sites in $\mathcal{R}_t$ and waits for the response from the sites in $\mathcal{R}_t$ (note that C knows all the sites in $\mathcal{R}_t$ based on list of $InitVal$s in $AcceptVal$). If any site responds with $Decision$ as True, this implies the previous leader was successful in making the value fault-tolerant but failed before sending Decision to all. Hence, site C sends the Decision message to all sites in $\mathcal{R}_t$.

Otherwise, if any site responds with $AcceptVal = \perp$, C can safely abort the redistribution (and perhaps notify other sites in $\mathcal{R}_t$) as this implies that the previous leader failed before making the constructed value fault-tolerant, and hence could have decided on it. If all sites in $\mathcal{R}_t$, except the previous leader S, respond with identical $AcceptVal$, this implies that S was successful in storing the value on all sites in $\mathcal{R}_t$ but failed before sending any Decision message. Hence, site C decides on that value, sets $Decision$ to True, and sends the Decision message. And finally, if C cannot communicate with all the other blocked sites in $\mathcal{R}_t$, C is blocked.

Fault tolerance: Similar to Avantan[$\frac{n+1}{2}$], failures during protocol execution can cause the set of sites, $\mathcal{R}_t$, participating in that execution of Avantan[$*$] to be blocked. But since Avantan[$*$] allows fewer number of sites to participate in a redistribution compared to Avantan[$\frac{n+1}{2}$], the set of sites not participating in the t^{th} instance of Avantan[$*$] are free to serve client requests or execute another instance of redistribution. The experiments in Section 5 analyze and contrast the fault tolerance of the two versions on Avantan in a practical setting.

Theorem 2*: No two distinct values are both chosen by the set of sites participating in a given instance of Avantan[$*$].*

A value is chosen once ***all*** the sites participating in an instance of redistribution, denoted by $\mathcal{R}_t$, respond with Accept-Ok messages. In Avantan[$*$], an individual site participates in only one redistribution at a time and rejects any other concurrent redistribution request. Due to this behaviour, in a failure-free execution, sites in $\mathcal{R}_t$ have a single leader who proposes a single value. Thus, in failure-

free executions, all sites participating in an instance of Avantan[$*$] agree on a single value.

If one or more sites fail while executing Avantan[$*$], the recovery mechanism indicates that the live sites can either successfully terminate the redistribution or are blocked until more sites recover. Blocking implies the sites will not participate in other redistributions until the current redistribution instance is terminated, and hence, the sites in $\mathcal{R}_t$ will not choose two distinct values for t^{th} instance. □

While Avantan seems similar to Paxos, they differ in two major ways: (i) Paxos aims to reach agreement on a single, client provided value whereas Avantan collects partial values from each site and aims to reach agreement on the aggregated values, and (ii) the redistribution correctness condition (Equation 1) does not require a majority – a fact that is exploited in designing Avantan[$*$]– which is stringent requirement of Paxos.

4.4. Reallocating Tokens

After a site triggers redistribution and a subset of the sites execute either versions of Avantan protocol successfully, the sites execute a deterministic procedure to reallocate the tokens. In this section, we discuss how to compute the spare tokens and the procedure to reallocate the spare tokens.

A successful execution of either versions of Avantan ensures agreement on the $AcceptVal$, which is a list of $InitVal$s, i.e.,:

$$\mathcal{L}_t =< e, TU_t, TL_t, TW_t > \forall\, i \,\epsilon\, \mathcal{R}_t \tag{6}$$

The reallocation logic defined in Algorithm 2 takes $\mathcal{L}_t$ as input and reallocates the available tokens among the set of sites in $\mathcal{R}_t$. *The redistribution algorithm ensures the constraint in Equation 1* that at no point does the token allocation count across all sites exceed the maximum limit $\mathcal{M}_e$ for a given entity e. For ease of exposition, we again focus of reallocating the tokens for a single entity e and use the variables defined in Table 2 to explain the algorithm.

Redistributing tokens: As defined in line 1 of Algorithm 2, the RedistributeTokens procedure takes $\mathcal{L}_t$ as input. The spare tokens and the total tokens wanted (sum of tokens specified in the $TokensWanted$ field of each site) across all sites in $\mathcal{R}_t$ are computed as shown in lines 4-6. If the spare tokens are more than the total tokens wanted, all pending client requests can be satisfied, and AllocateTokens procedure is called.

Rejecting requests: If the tokens wanted is more than the spare, some requests must be rejected. The logic for handling this case is defined in the procedure at line 10 of Algorithm 2. We take a greedy approach to *maximise overall token usage* rather than maximise the number of requests satisfied. This is achieved by first sorting the list $\mathcal{L}_t$ in ascending order of tokens wanted (line 11); we choose ascending order since the algorithm can reject requests with least tokens wanted first. From this ascending ordered list, requests with smaller number of tokens wanted are rejected (by setting tokens wanted to 0 in line 13 and increasing the spare quantity in line 14) until the number of spare tokens exceed total tokens wanted (lines 15-16).

Algorithm 2 Procedures to re-allocate spare tokens after a successful redistribution

1: **procedure** REDISTRIBUTETOKENS($\mathcal{L}_t$)
2: $\mathcal{S}_t \leftarrow 0$ /* Spare tokens */
3: $TotalTW \leftarrow 0$ /* Total tokens wanted*/
4: **for** i in $\mathcal{R}_t$ **do**
5: $TotalTW \leftarrow TotalTW + \mathcal{L}_t[i].TW_t$
6: $\mathcal{S}_t \leftarrow \mathcal{S}_t + \mathcal{L}_t[i].TL_t$
7: **if** $TotalTW > \mathcal{S}_t$ **then**
8: $\mathcal{L}_t, \mathcal{S}_t \leftarrow$ RejectSomeRequests($\mathcal{L}_t, \mathcal{S}_t$)
9: AllocateTokens($\mathcal{L}_t, \mathcal{S}_t$)
10: **procedure** REJECTSOMEREQUESTS($\mathcal{L}_t, \mathcal{S}_t$)
11: $sorted\mathcal{L}_t \leftarrow \mathcal{L}_t$ sorted in ascending order of TW_t
12: **for** i in $sorted\mathcal{L}_t$ **do**
13: $sorted\mathcal{L}_t[i].TW_t \leftarrow 0$
14: $\mathcal{S}_t \leftarrow \mathcal{S}_t + sorted\mathcal{L}_t[i].TL_t$
15: **if** $TotalTW \leq \mathcal{S}_t$ **then**
16: break
17: **return** $sorted\mathcal{L}_t, \mathcal{S}_t$
18: **procedure** ALLOCATETOKENS($\mathcal{L}_t, \mathcal{S}_t$)
19: **for** i in $\mathcal{R}_t$ **do**
20: $\mathcal{L}_t[i].TokensGranted \leftarrow \mathcal{L}_t[i].TW_t$
21: $\mathcal{S}_t \leftarrow \mathcal{S}_t - \mathcal{L}_t[i].TW_t$
22: **for** i in $\mathcal{R}_t$ **do**
23: $\mathcal{L}_t[i].TokensGranted \leftarrow \mathcal{L}_t[i].TokensGranted + \frac{\mathcal{S}_t}{len(\mathcal{R}_t)}$

Allocating spare tokens: Finally, AllocateTokens (line 18) is called with updated list $\mathcal{L}_t$ and spare tokens $\mathcal{S}_t$. At this point, the redistribution satisfies all sites with non-zero tokens wanted (as the requests that cannot be satisfied are already rejected). A tokens request is granted as shown in line 20 and for each granted request, the spare quantity is updated (line 21) After satisfying all the tokens wanted requests, if any more tokens are left, they are equally distributed among all the participating sites (line 23).

5. Experimental Evaluation

In this section we discuss the experimental evaluation of Samya, specifically the performance of two versions of Samya where one version uses Avantan[$\frac{n+1}{2}$] and the other uses Avantan[$*$] to handle any redistributions during the experiments. Samya's performance is compared with two baselines implemented by us in Go and one open-sourced database:

i). *MultiPaxSys*: A Spanner-like geo-distributed database that executes multi-Paxos [11] for each transaction.

ii). *Demarcation/Escrow*: A value-partitioned system that captures the underlying mechanisms proposed in [10], [25], [9], [24]. Specifically, Demarcation/Escrow extends the Demarcation protocol proposed by Barbara et al. [10] to more than 2 sites, similar to [9] by Alonso et al., and integrates the notion of site escrows used in [25] by Kumar et al. All sites start with an equal 'escrow' of an entity e (maximum limit, $\mathcal{M}_e$, divided equally among all sites), and the sites serve requests locally until they exhaust the spare escrow locally. When a request cannot be served locally at site i, i borrows escrows from one or more sites. If site i fails to borrow escrow from other sites (if they are also out of escrows), then site i rejects the client request. A stringent requirement of this baseline, inherited from [10] and [25], is it requires the network to be reliable; a message drop may lead to blocking.

(iii). *CockroachDB*: A state-of-the-art open sourced geo-distributed database that uses Raft[32] to replicate any changes to the data.

In evaluating Samya, the experiments focus on two performance aspects: *commit latency* – time taken to commit a transaction measured by the client as the time from when it sent a transaction to when it received a response to that transaction; and *throughput* – the number of transactions successfully committed per second, i.e, only the *acquireTokens* and *releaseTokens* requests that succeed are counted in throughput.

5.1. Resource Demand Data and Its Prediction

Samya is evaluated on a VM workload dataset published by Microsoft Azure [3]. The dataset, consisting of roughly 2 million data points, contains a representative trace of Azure's VM workload in a single geographical region collected over a month in 2017. Along with other information, it includes VM creation and deletion requests reported at discrete 5-minute intervals. A detailed description and an analysis of the dataset is published by Cortez et al. [14] where several interesting patterns of the dataset are demonstrated. A noteworthy observation among them is that the VM requests have nearly periodic properties over time. The authors conclude that for such requests, "history is an accurate predictor of future behavior". We leverage the periodic property of the requests in the dataset to build a prediction module for Samya.

5.1.1. Resource Demand Prediction. The original Azure data was pre-processed such that the number of VM creations and deletions represent a demand for VMs, as depicted in Figure 7. The figure shows the periodically increasing and decreasing demand patterns in the data, indicating that a learning model can learn these patterns to predict future demands. Although the cloud computing literature consists of many sophisticated learning methods for resource prediction, we picked 3 simple options for resource prediction: the random walk model as the baseline model, ARIMA (autoregressive integrated moving average) as a linear regression model, and LSTM, a type of recurrent neural network, as a non-linear regression model.

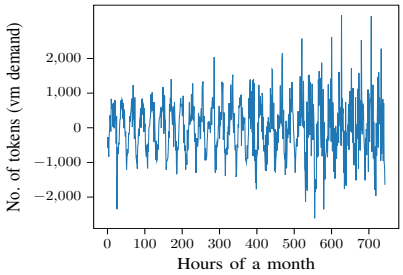

Figure 7: Resource demand data recorded for one month at a single region.

To evaluate which out of the three models best predict the VM demands in Azure dataset, the original one month data was split into 80% of training data and 20% of testing data. The result of our evaluation is shown in Table 4. LSTM predicted the resource demands with highest accuracy, and hence, was chosen as the prediction module for Samya.

TABLE 4: Mean Absolute Error (MAE) - in units of number of tokens - of resource demand prediction for three different prediction models.

	Random Walk	ARIMA	LSTM
MAE (no. of tokens)	1212.19	609.13	259.21

5.1.2. Data processing. Since Samya is proposed as a solution for the *hot-spot* problem of aggregate data, the dataset used to evaluate Samya needs to have a high request-arrival-rate. To achieve this, we modified the original data's sampling interval of 5 minutes to 5 seconds. As a result, the same number of requests that arrived in a span of 5 minutes in the original dataset now arrived in a span of 5 seconds, generating a workload with high request-arrival-rate. Due to the shrinking of the sampling interval, the original duration of 30 days of the entire dataset was reduced to 12 hours.

From this 12 hours of data, we trained the LSTM prediction model with 11 hours of data, and used the last one hour (corresponding to 60 hours in the original dataset) to generate client transactions. This train and test ratio differs from the 80:20 ratio used specifically to evaluate various prediction models and to choose one among them; for real-deployment, the train and test ratio was changed to approximately 90:10.

Samya is a geo-distributed system with sites across different time zones while the Azure dataset corresponds to only a single geographical region in a single time zone. To generate the client requests at different regions, the original dataset is phase shifted based on the time difference between the regions. For example, if the demand in the original dataset peaks at 10 AM Tuesday and drops at 1 AM Wednesday, in our experiments, clients in North America generate peak demand load at 10 AM Tuesday *at the same time* as clients in Asia generate the reduced demand of 1 AM Wednesday – phase-shifted demands corresponding to

the time difference between North America and Asia. The phase shifting retains the periodicity in each region while avoiding peak demand in all regions at the same time. The clients in different regions generate respective phase-shifted transactional workloads where the VM creation and deletion requests from the dataset are transformed to *acquireTokens(VM, 1)* and *releaseTokens(VM, 1)* requests respectively.

5.2. Experimental Setup

The three systems, Samya, Demarcation/Escrow, and MultiPaxSys were deployed on Google Cloud Platform where each server was a general purpose n1-standard VM with 8 vCPUs and 30 GiB RAM. For most experiments, the VMs were placed in 5 different regions: US-West1 (US), Asia-East2 (AS), Europe-West2 (EU), Australia-Southeast1 (AU), and SouthAmerica-East1 (SA). 3 to 5 is the typical default number of replicas used in current state-of-the-art databases [4], [5]. The inter-region latency is presented in Table 5.

TABLE 5: Inter-region latencies in ms.

	AS	EU	AU	SA
US	131	132	161	180
AS	-	262	125	302
EU	-	-	265	218
AU	-	-	-	305

To simplify the evaluation, in the experiments, we merged the application managers and clients into a single machine. Thus, each region consisted of one VM as the client generating token acquire or release requests and another VM as the server serving client requests. In the experiments, *all five clients generated phase-shifted transactions simultaneously* and a client's requests were served by the site closest to it.

For MultiPaxSys and CockroachDB, since the recommendation is to place a majority of the sites in close-by regions to achieve faster replication time, we placed 3 out of 5 sites in different regions within the US, and 2 others in Asia and Europe.

All the experiments focused on entity VM and the maximum global limit, $\mathcal{M}_e$, was set to 5000, indicating that each site in Samya and Demarcation/Escrow starts with 1000 tokens. Note from Figure 7 that a single region's demand can go beyond 1000, ensuring that sites in Samya would require redistribution. Another implementation specific optimization was when to perform predictions: since prediction can be computationally expensive, in our experiments, a site predicts future demand only when its $TokensLeft$ value is 20% of the tokens granted value in the previous redistribution round. If the prediction indicates an increase in demand, the site triggers a proactive redistribution.

5.3. Latency and throughput

The first set of experiments evaluate the commit latency and throughput of the two versions of Samya, and

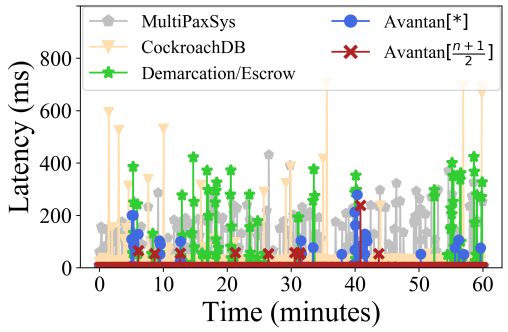

Figure 8: Latency of each transaction sent for an hour.

the three baselines: Demarcation/Escrow, MultiPaxSys and CockroachDB by generating load for one hour (corresponding to 60 hours in the original dataset), creating roughly 820000 transactions. The goal of this experiment is to study the behavior of the systems over extended periods of time when the workload is highly contentious (each request either acquires or releases tokens for the same entity, VMs).

Latency: Figure 8 plots a sample of commit latencies of *individual transactions* sent by clients for the duration of an hour. Since each transaction in MultiPaxSys and CockroachDB executes a replication round before responding to the client, and the workload is contentious, both the systems incur significantly higher latencies compared to Samya. For Samya (both versions), most client requests are served locally at the closest site; the spikes in latencies of specific transactions indicate an ongoing redistribution during that transaction's processing. For Demarcation/Escrow, although most requests are served locally, due to the lack of prediction and an efficient escrow redistribution strategy, the peaks in resource demand causes latency peaks; hence latency of Demarcation/Escrow is higher than Samya.

Latency incurred at different percentiles for all five systems are tabulated in Table 6. The interesting behaviour here is the contrast in latency numbers for Avantan[$*$] and Avantan[$\frac{n+1}{2}$]. We suspected Avantan[$*$] to outperform Avantan[$\frac{n+1}{2}$], since the latter needs to wait for responses from a majority to execute a redistribution, unlike Avantan[$*$], which can proceed with any number of responses. But the latencies in Table 6 indicate the opposite – Avantan[$\frac{n+1}{2}$] has lower latencies than Avantan[$*$] across all percentiles.

This counter-intuitive result is explained by the difference in how the two versions construct the value during the first phase of the redistribution protocol. Avantan[$\frac{n+1}{2}$] requires at least a majority of sites to respond with their local token values, which the leader concatenates into a single value (i.e., $AcceptVal$). This redistribution re-balances the tokens between a majority of sites. Whereas, Avantan[$*$] collects just enough responses (consisting of local token values) to satisfy its token needs, and immediately proceeds to the fault-tolerance phase. While this greedy approach may be beneficial for specific transactions, Avantan[$*$] ends up re-balancing the tokens between a small number of sites, causing more sites to trigger subsequent redistributions. Hence,

TABLE 6: Various latency percentiles in ms.

	Samya w/ Av.[$\frac{n+1}{2}$]	Samya w/ Av.[$*$]	Demarcation/ Escrow	MultiPaxSys	CockroachDB
90^{th} percentile	1.40 ms	2.9 ms	3.5 ms	126.8 ms	158.7 ms
95^{th} percentile	10.2 ms	37.3 ms	59.6 ms	172.7 ms	184.2 ms
99^{th} percentile	65.1 ms	97.3 ms	213.9 ms	276.3 ms	351.4 ms

in the long run, Avantan[$\frac{n+1}{2}$] is better at re-balancing the tokens and causing fewer redistributions. In the experiments, for the same client workload, Avantan[$\frac{n+1}{2}$] required 208 redistributions (proactive and reactive combined) whereas Avantan[$*$] required 792 redistributions.

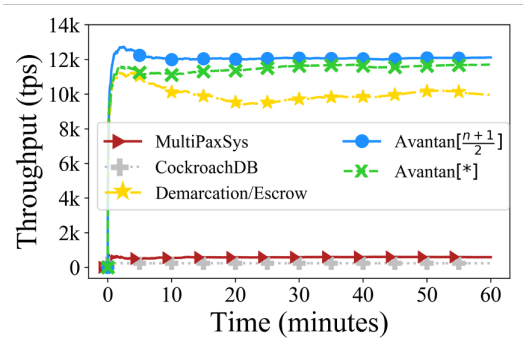

Figure 9: Throughput of the systems recorded for an hour.

Throughput: Figure 9 shows the 5-minute moving average throughput of all five systems when five clients generate concurrent requests each second and send the requests to the sites. Since MultiPaxSys and CockroachDB serve these requests sequentially (as they all update the same data entry), their throughput is roughly **16-18x** worse than Samya and **11x** worse that Demarcation/Escrow. This result highlights the *benefits of dis-aggregating an aggregate value to allow executing concurrent transactions*.

Between Demarcation/Escrow and Samya, the demand prediction and a more efficient redistribution strategy of Samya causes its throughput to be almost **1.3x** better than Demarcation/Escrow. The performance difference between Avantan[$\frac{n+1}{2}$] and Avantan[$*$] is due to the increased number of redistribution in the latter, which slows the rate with which client requests are served.

Since this experiment establishes that the performance of MultiPaxSys and CockroachDB are comparable, we use MultiPaxSys for performance comparisons in the following experiments.

5.4. Failure Experiments

5.4.1. Crash Failures. This set of experiments evaluate the Samya and MultiPaxSys when crash failures occur (Demarcation/Escrow is not evaluated in failure experiments for it requires reliable networks and hence is not fault-tolerant). The experiment starts with five regions and roughly every 10 minutes, we crash both the site and the client in a region, until only one region remains alive, while recording the throughput throughout the experiment. The results are

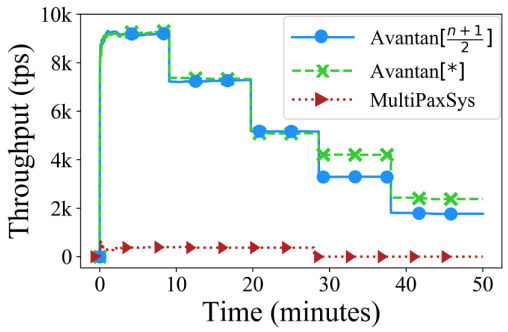

Figure 10: Throughput recorded when sites fail.

highlighted in Figure 10. As indicated in the figure, once three sites crash, the throughput of MultiPaxSys drops to 0, since no transaction can be committed once a majority of the sites fail.

For the two versions of Samya, the performance is roughly the same up to 2 site failures (note that the performance is similar for both and not worse for Avantan[$*$] because in the first few minutes, the number of redistributions are low due to low resource demand in the Azure dataset; when the number of redistributions are low, the two versions perform comparably). When 3 sites fail, Avantan[$\frac{n+1}{2}$] attempts redistribution, times-out, and fails to perform any redistribution due to the failed majority. However, sites continue to serve requests that can be served locally. Meanwhile, Avantan[$*$] can successfully redistribute tokens even if only a minority of the sites are alive, thus causing its performance to be higher than Avantan[$\frac{n+1}{2}$] when failures occur.

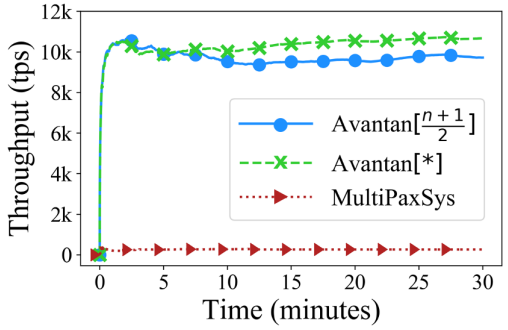

Figure 11: Throughput recorded during network partition.

5.4.2. Network Partitions. This experiment measures the performance of Samya and MultiPaxSys during a network

partition. The experiment is performed in the presence of a 3-2 network partition, i.e., one partition consists of 3 sites and the other consists of 2 sites, and clients send transactions for thirty minutes. The results are indicated in Figure 11. In MultiPaxSys, only the partition with 3 replicas continues to serve client requests and are up-to-data while the other two replicas are rendered stale. Its performance is significantly low compared to Samya.

For Samya, although both Avantan[$\frac{n+1}{2}$] and Avantan[$*$] start off with comparable performance, once the sites exhaust local tokens and trigger redistributions, Avantan[$*$] outperforms Avantan[$\frac{n+1}{2}$], since Avantan[$\frac{n+1}{2}$] cannot redistribute tokens in the smaller network partition whereas Avantan[$*$] can.

The two failure experiments highlight that between the two versions of redistribution strategies for Samya, Avantan[$*$] performs better in a failure prone environment, compared to Avantan[$\frac{n+1}{2}$]; but in a failure-free scenario, Avantan[$\frac{n+1}{2}$] performs better as indicated in Section 5.3.

One advantage of MultiPaxSys over Samya in both failure scenarios is that MultiPaxSys can allot more tokens as long as a majority of the replicas are alive, because the synchronous replication makes sure that the entire quota limit can be used. Whereas some tokens claimed tokens in Samya are lost temporarily until recovery.

5.5. No Constraint vs. No Redistribution

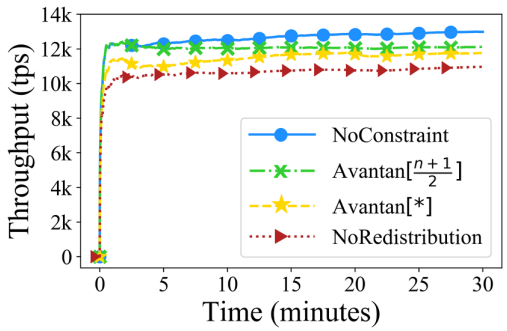

Figure 12: Throughput of Samya with no constraints and no redistributions vs. Samya with redistributions.

The remaining experiments focus on contrasting the two version of Avantan in Samya. In this experiment, we explore whether redistribution is worth it and the cost of redistribution on throughput. This experiment compares Samya's performance with its two baseline versions: i). *No Constraints*: there is no upper-bound on the number of resource tokens allotted, hence every requests (acquire or release) succeeds locally at a site; ii). *No Redistribution*: there is a maximum limit constraint but once a site exhausts its local quota, it simply rejects the client request, rather than triggering a redistribution (neither proactive nor reactive). The results are shown in Figure 12.

Comparing the baselines: i). Samya with no constraints is the best case scenario with optimal performance, and as seen in Figure 12, Samya with constraints and redistributions has only 3.5-4% less throughput than the optimal throughput. ii). Samya with both versions of Avantan has about 14% higher throughput than Samya with no redistributions, i.e., 14% of the transactions would be rejected if Samya did not perform redistributions. This indicates that although executing global redistribution is expensive, the system performs better with the redistributions.

5.6. Proactive vs. Reactive Redistributions

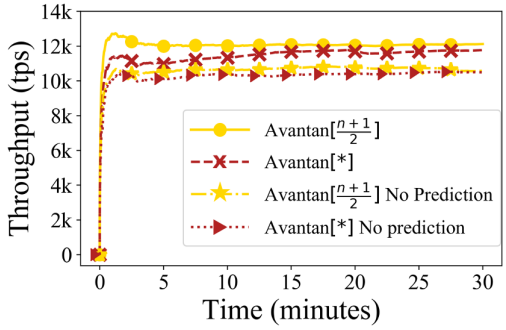

Figure 13: Samya's performance with and without predictions.

This experiment aims to measure the significance of predictions in Samya. Performance of four variants of Samya are measured: Avantan[$\frac{n+1}{2}$] with and without prediction, and Avantan[$*$] with and without prediction. The clients execute transactions for thirty minutes for each variant. As indicated in Figure 13, Samya performs about **1.4x** better with predictions (for both versions). Predictions proactively prepare a site for the incoming demand and allows a site to indicate its token requirements with higher precision. This experiment highlights the advantages of using predictions in building distributed systems such as Samya.

5.7. Increasing number of sites

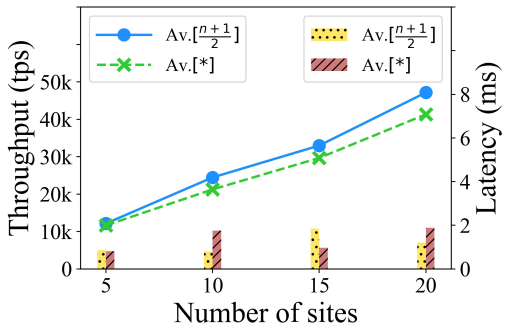

Figure 14: Average throughput (line graphs) and latency (bar graphs) measured for increasing number of sites.

This set of experiments evaluate the scalability of Samya by increasing the number of sites from 5 to 20, with addi-

tional sites spawned in each of the 5 regions in which previous experiments were conducted. In this experiment, for each configuration, the clients generate transactions for 10 minutes. Figure 14 depicts the average latency and average throughput for each configuration. As indicated in the figure, Samya shows a roughly linear increase in throughput as the number of sites increase, while keeping the average latency below 2ms for both versions of Avantan. This experiment highlights that Samya is highly scalable as more clients can concurrently acquire or release tokens when the number of sites increase.

5.8. Read-Write workload

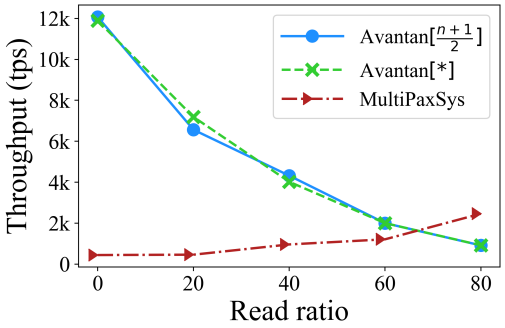

Figure 15: Average throughput measured with increasing ratio of read-only transactions.

This experiment compares the average throughput of the two versions of Avantan with that of MultiPaxSys, when the ratio of read-only transactions increases, as shown in Figure 15. For Avantan, when a client issues a read request to a site S, S communicates with all the other sites to learn their current token availability, aggregates the received values and responds to the client with a global snapshot of the total available tokens. For MultiPaxSys, the current available tokens is read at a single leader site. This experiment highlights the threshold at which MultiPaxSys has performance advantages over Avantan: when the read ratio increases roughly past 65%, the throughput of MultiPaxSys increases more than Avantan. Since reads are performed at a single site in MultiPaxSys and most writes are performed a single site in Samya, one would expect the crossover point to be at 50%, which is not the case. The reason is: in our experimental setup, five geo-distributed clients generate requests in parallel and for MultiPaxSys, all client requests are sent to one single leader site, which sequentially processes the requests, thus incurring high latency. Whereas for Avantan, due to the decentralised design choice, write requests are typically served locally by sites closest to the clients, in parallel. Hence, as long as an application's write load is 35% or more, it can benefit by choosing Samya.

5.9. Varying the maximum limit $\mathcal{M}_e$

This experiment compares the average throughput of the two versions of Avantan when the maximum limit $\mathcal{M}_e$ of

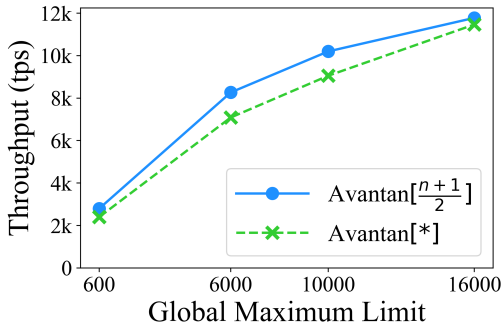

Figure 16: Average throughput measured with increasing maximum resource limit $\mathcal{M}_e$.

VM resource increases from 600 to 16000, as shown in Figure 16. This experiment consists of 5 sites in 5 different regions and each experimental run was executed for half an hour. From the VM demand data shown in Figure 7, 624 is the mean positive demand and 3118 is max positive demand at a *single* region. Hence, in this experiment we set the maximum limit from 600 to 16000. When $\mathcal{M}_e$ is set to 600, each site starts with 120 tokens each, causing roughly 1960 redistributions (predictive and reactive combined); similarly, with $\mathcal{M}_e$ set to 16000, each site starts with 3200 tokens causing no redistribution. The experiment shows that Avantan's throughput increases roughly **5x** when the maximum limit is increased from mean to max demand for the specific Azure VM demand data, thus bringing out the sensitivity of Avantan's performance with regard to the maximum resource limit.

5.10. Request arrival rate

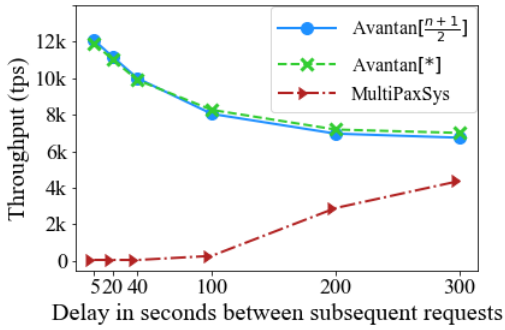

Figure 17: Average throughput measured with increasing delay between requests.

This experiment measures the sensitivity of Avantan to the request arrival rate. As mentioned in Section 5.1.2, to generate a high request arrival rate, the original data's sampling interval was modified from 5 minutes (300 seconds) to 5 seconds. This experiment starts with 5 second interval and goes up to the original scale of 300 seconds. For each configuration, we measure the average throughput of a 5-minute moving average throughput. As seen in Figure 17, the throughput of Avantan reduces by 33% when the request

arrival rate reduces by 60x (5 seconds to 300 seconds). For MultiPaxSys, we notice as increase in throughput for reduced request arrival rate as the number of contentious requests sent per second reduces, causing MultiPaxSys to commit more transactions (i.e., the rate of aborted transactions decreases). The main conclusion of this experiment is that even at the original request arrival rate, Avantan commits 43% more transactions than MultiPaxSys.

5.11. Limitations and Future Work

Effect of Maximum Limit $\mathcal{M}_e$: Samya's performance is inversely correlated to the number of redistributions executed by the sites (as indicated in Section 5.3). The higher the maximum limit, $\mathcal{M}_e$, for a resource e, the fewer the number of redistributions sites need to execute. Hence, if most resources of an application have small maximum limit value, Samya's performance benefits may reduce, as indicated in the experiment presented in Section 5.9.

Read-Heavy Workloads: In its current design, Samya is optimized for update heavy workloads. Samya can be easily extended to incorporate partial read transactions, i.e., transactions that read partial resource availability information stored at a single site. To obtain global resource availability information, sites in Samya execute a round of communication, unlike reading at the leader in Spanner-like systems. Therefore, Samya is a better choice if the client workload consists of at least 35% writes, as indicated in Section 5.8.

Global Predictions: In its current design, a site in Samya predicts future demand locally and triggers a redistribution. In addition to relying on each site's local knowledge to determine when to trigger redistribution, a global optimizer can be designed that predicts workload spike/trough at each site and triggers redistribution based on the global knowledge. This is an interesting future direction.

6. Conclusion

In this paper, we propose *Samya* – a geo-distributed data management system to store aggregate data, presented as a system that specifically maintains cloud resource usage data. Samya dis-aggregates the aggregate resource usage data and stores fractions of available tokens of resources on multiple geo-distributed sites. The dis-aggregation allows concurrent updates to the hotspot data, in contrast to sequentially ordering all concurrent and contentious updates at a leader site as in traditional geo-distributed databases such as Google's Spanner. A site in Samya serves client requests independently until, based on a learning mechanism, it predicts an increase in its local resource demand that cannot be satisfied locally. This triggers a synchronization protocol *Avantan* to redistribute the available tokens, after which, sites continue to serve client requests independently. We discuss two version of Avantan where one version performs better in an infrequent failure setting, and the other performs better when crash failures or network partitions are frequent. The experimental evaluation of Samya's performance highlights the benefit of dis-aggregation as Samya commits 16x to 18x more transactions than a Spanner-like database.

Acknowledgements

Sujaya Maiyya is partially supported by IBM PhD Fellowship. This work is funded in part by NSF grants CNS-1703560 and CNS-1815733.

References

[1] Amazon AWS Account Hierarchy. https://aws.amazon.com/answers/account-management/aws-multi-account-billing-strategy/. Accessed: 2020-02-17.

[2] Azure CosmosDB. https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/. Accessed: 2020-06-17.

[3] Azure Public Dataset. "https://github.com/Azure/AzurePublicDataset". 2019 (accessed June 30, 2020).

[4] Default Replica Count For CockroahDB. "https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html". Accessed Jan 10, 2021).

[5] Default Replica Count For Spanner. "https://cloud.google.com/spanner/docs/instances". Accessed Jan 10, 2021).

[6] Google Cloud Enterprise Hierarchy. https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations. Accessed: 2020-02-17.

[7] Microsoft Azure Resource Hierarchy. https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-setup-guide/organize-resources?tabs=AzureManagmentGroupsAndHierarchy. Accessed: 2020-02-17.

[8] Resource Tracking Services. "https://thedigitalprojectmanager.com/resource-scheduling-software-tools/". Accessed Oct 3, 2020).

[9] G. Alonso and A. El Abbadi. Partitioned data objects in distributed databases. *Distributed and Parallel Databases*, 3(1):5–35, 1995.

[10] D. Barbara and H. Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *International Conference on Extending Database Technology*, pages 373–388. Springer, 1992.

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[12] Z. Chen, Y. Zhu, Y. Di, and S. Feng. Self-adaptive prediction of cloud resource demands using ensemble model and subtractive-fuzzy clustering based fuzzy neural network. *Computational intelligence and neuroscience*, 2015, 2015.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[14] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 153–167, 2017.

[15] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.

[16] 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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[17] S. Di, D. Kondo, and W. Cirne. Host load prediction in a google compute cloud with a bayesian model. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[19] L. Golubchik and A. Thomasian. Token allocation in distributed systems. In *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 64–71. IEEE, 1992.

[20] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

[21] T. Härder. Handling hot spot data in db-sharing systems. *Information Systems*, 13(2):155–166, 1988.

[22] Y. Jiang, C.-s. Perng, T. Li, and R. Chang. Asap: A self-adaptive prediction system for instant cloud resource demand provisioning. In *2011 IEEE 11th International Conference on Data Mining*, pages 1104–1109. IEEE, 2011.

[23] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126, 2013.

[24] N. Krishnakumar and A. J. Bernstein. High throughput escrow algorithms for replicated databases. In *VLDB*, volume 1992, pages 175–186, 1992.

[25] A. Kumar and M. Stonebraker. Semantics based transaction management techniques for replicated data. *ACM SIGMOD Record*, 17(3):117–125, 1988.

[26] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[27] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, 2013.

[28] S. Maiyya, F. Nawab, D. Agrawal, and A. E. Abbadi. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment*, 12(5):611–623, 2019.

[29] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 517–532, 2016.

[30] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. {AGILE}: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 69–82, 2013.

[31] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[32] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14)*, pages 305–319, 2014.

[33] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[34] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[35] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[36] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[37] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[38] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[39] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 511–526, 2020.

[40] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–37, 2018.