Consistify: Preserving Correctness and SLA Under Weak Consistency

Subhajit Sidhanta
Indian Institute of Technology
Jodhpur
Jodhpur, India
subhajit@iitj.ac.in

Supratik Mukhopadhyay
Louisiana State University
Baton Rouge, USA
supratik@csc.lsu.edu

Wojciech Golab
University of Waterloo
Waterloo, Canada
wgolab@uwaterloo.ca

ABSTRACT

While executing a client application, comprising a sequence of one or more storage operations (read, write, etc.), replicated datastores provide the option of tuning per-operation client-centric consistency settings. Weaker consistency settings allow client applications to execute with lower observed latency due to less coordination among the replica, improving throughput. To enable a client application running on a geo-replicated datastore to meet stringent *SLA* (Service Level Agreement) *deadlines*, i.e., latency thresholds provided in the *SLA*, users apply weak client-centric consistency settings. Strong consistency settings may result in higher latency due to coordination among greater number of replicas. However, since a weak consistency setting requires only a partial quorum of replicas to coordinate, some of the replicas may be overwritten by conflicting updates from concurrent clients, which, in turn, may result in violation of *correctness conditions* specified by the developer. Such correctness conditions are provided as *postconditions* that impose restrictions on the values observed by the clients. Enforcing correctness conditions under weak consistency settings on geo-replicated datastores from the client application code is a tedious and challenging task. The task is even more complicated when considering application-level *SLA* deadlines. Deadline (latency) is one of the most important *SLA* parameters for optimizing client-centric performance. To this end, we present Consistify, a novel framework, that acts as an interface layer between client applications and underlying datastores, and enables client applications to execute with weak per-operation consistency settings, while simultaneously respecting: 1) the correctness conditions, specified using logical predicates, and 2) the *SLA* deadline. Using benchmark workloads, we experimentally demonstrate that Consistify outperforms the state-of-the-art systems, namely QUELA. For several other systems that provide flexible consistency

settings, namely Bolt-on causal system, SIEVE, and Cure, we provide a comparison of the throughput and latency observed with Consistify against the numbers reported with those systems in prior papers.

CCS CONCEPTS

• Information systems → Storage management;

1 INTRODUCTION

In geo-replicated datastores [11, 14, 16, 21], weaker consistency settings typically produce low observed latency, since they require fewer replica nodes to coordinate before returning the result of a given operation; and thus can help improve throughput. To enable a client application to meet stringent *SLA* deadlines, geo-replicated datastores provide users the choice of running applications with weak client-centric consistency settings, i.e., consistency from the viewpoint of a client application [16]. However, under a weak client-centric consistency setting, a client application runs the risk of observing incorrect results, i.e., results that do not conform to the business logic of an application, due to conflicting updates from concurrent clients that may overwrite the replicas. Lu et al. [20] describes a potential example of such incorrect application behavior in Facebook. In their example, under weak client-centric consistency settings, users of social networking applications, like Facebook, can observe out-of-order comments to blog posts (discussed in Section 2).

We address the problem of enforcing given correctness conditions, while applying weak per-operation consistency settings for improving client-centric performance and meeting *SLA* deadlines of client applications. This can be achieved in one of the following ways: Approach 1) by manually modifying the application code, adding code to introduce extensive application-level checks for correctness violations, Approach 2) by developing a prototype distributed datastore or modifying an existing datastore such that the said datastore internally can adapt its configurations and the manner in which the datastore handles storage operations to achieve a given *SLA* deadline and preserve given correctness conditions, or Approach 3) by developing a middleware that automates the tuning of per-operation consistency settings of the underlying datastore, and controls execution order of operations comprising the application by maintaining a queue of operations which allows execution order to be altered and buffers operations according to the dependency relationship between operations. Approach 1 can result in increasing size

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICDCN '19, January 4–7, 2019, Navi Mumbai, India

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6094-4/19/01...\$15.00

<https://doi.org/10.1145/3288599.3288630>

and complexity of application code [8, 19]. Approach 2 requires developing a new prototype datastore from scratch, or modifying an existing datastore, which in turn, requires extensive development effort as well as considerable understanding of the codebase of the datastore [3, 6].

For maintaining correctness conditions, like ordering of results in web applications, e.g., the ordering of blog comments in Facebook, researchers have followed Approach 3, and developed frameworks that can upgrade existing datastores to provide causal consistency [8, 19]. Such systems preserve causality by implementing dependency tracking mechanisms through metadata of datastore operations, and by maintaining local snapshots of the datastore to allow operations to execute out of order without violating dependency relationships according to the causality property. As discussed in [8], maintaining local snapshots can be expensive for modern production scale web applications, like Facebook. Further, according to [8, 19], these systems are often unable to simultaneously preserve “convergent causality” across concurrent applications, and also provide high application throughput under heavy concurrent workload.

To address the above issues, we present Consistify, a middleware that acts as an interface between a client application that achieves the above goals by tuning the per-operation consistency settings of the datastore. The open source codebase for the Java-based implementation of the Consistify interface can be found in [28]. For applications that use a relational data model to process data, reordering of some of the storage operations comprised in an application does not violate application level correctness [25]. Further, most transaction processing systems allow aggressive reordering of storage operations within scope of the transaction. Applications which implement a high level abstract data type called conflict-free replicated data type (CRDT) [26] allow unrestricted reordering of operations. Consistify provides mechanisms that allow selective reordering of operations comprised in applications which do not leverage CRDTs, executing some operations in parallel while executing the rest in a blocking manner. Consistify tunes a client application with weak per-operation consistency settings, to improve application throughput. At the same time, Consistify prevents violation of given correctness conditions by controlling the execution order of client applications (refer to Section 3), and by using metadata associated with the variables processed by the storage operations comprising the client applications. The above metadata is maintained using temporary control variables called *control locks*. We colocate control locks with the datastore objects, Consistify piggybacks the read/write requests for control flags on top of the datastore queries itself. Thus, maintaining control locks does not incur substantial communication overhead. Further, accessing a control lock does not require accessing the client application that invoked the operation which is associated with the control lock. Hence, accessing a control lock does not require any inter-process communication between the Consistify interface nodes. Consistify also comprises a Failure Handling module that resolves failure cases, such as if the

application fails to satisfy SLA deadlines, or if the application crashes without releasing a control lock. The rest of the paper presents the design of the framework and the experimental results in details. For lack of space, we have included an extended version of the paper in github [28].

1.1 Target System: Assumptions

To render our solution implementation-independent, we do not make any assumption regarding the internal protocol-level details of the target datastore, like the replication and partitioning scheme. We only assume that the target datastore is a geo-replicated datastore [13, 16] running on a cluster of nodes, which may be geographically distributed. In geo-replicated datastores, objects are internally stored as simple data types, namely key-value pairs. A key-value pair is also equivalent to a row of a table in database terminology. A key is the datastore-equivalent of the primary key of a relational database. Unlike relational database systems, the above data model supports only the basic read and write operations, i.e., read the value stored against a key, and update the value stored against a key, respectively.

A client application, executing on such a datastore, comprises a sequence of such read and write operations, that we call *queries*. We only consider a finite sequence of queries because our solution requires an instruction buffer of finite size as well as a finite space in the datastore for processing metadata in the form of control locks. For a client application Cl , we denote the ordered sequence of queries by Q_{Cl} , with the i th query in the sequence being denoted by q_i . Our approach is a static analysis approach where the sequence of queries in the given application is analyzed before executing the application on the datastore. Client applications that involve database transactions [3–6] usually comprise a static (i.e., a fixed) sequence of queries. Hence, our solution works only in those cases where the sequence of queries is known beforehand, and the above sequence does not dynamically change. A given consistency level C^i applied to a query q_i specifies the number f_i of replica nodes that must respond with acknowledgement for the query q_i . A consistency level C^i is considered weaker than another consistency level C^j , if C^i requires a smaller number of replicas to respond to than C^j . Thus, C^i specifies that $\lceil f_i \times 1/R \rceil$ replicas must acknowledge for q_i , where R denotes the applied replication factor, and f_i varies from 1 for consistency level $C^i = \text{ONE}$ to R for $C^i = \text{ALL}$.

2 BACKGROUND AND MOTIVATION

We target web services and applications that typically experience heavy volumes of concurrent workload. User bases of such applications often range up to millions. In such applications, concurrent users perform conflicting update operations on replicas of data objects stored in replicated storage systems, which may result in concurrency anomalies. Further, such applications are typically accessed by users that belong to different user groups. Different user groups have different roles, i.e., they perform different set of storage operations on

the underlying storage systems, and they have different levels of constraints on the processed data. Under heavy levels of concurrency in the application workload, users from different user groups concurrently process the same data objects. While a user from a user group with strict constraint on the processed data performs operations on given data objects, users from a different user group may concurrently overwrite the same data object. This may result in potential violation of constraints for users from the former user group. In the current state-of-the-art, such constraints have to be programmatically enforced from the application code. To relieve the application programmers from the burden, we present a middleware solution that interface between the client application and underlying storage systems.

Users can specify constraints on the data processed in such applications as a high level input parameter to the middleware, which we refer to as correctness conditions on the application code. Correctness conditions for such a client application Cl impose constraints on the values returned by the storage operations comprising Cl , like preventing the loss of items added in a shopping cart, or forbidding the presence of duplicate values for a unique data item. State-of-the-art systems [12], such as Quela [29, 30], specify application level correctness conditions using special-purpose declarative languages. Specifications in such languages are usually based on relations like visibility and session order which are defined in terms of database specific terminologies and may not be well known among general users. Contrastingly, Consistify accepts the correctness conditions for a given client application in the form of simple logical predicates.

```

If(role == user)
  if(action == add)
    add item to cart;
  if(action == delete)
    delete item from cart;
if(role == admin)
  visit website of the retailer for each item in the cart;
  Finalize/approve items in the cart;
  Finish purchase;

```

Figure 1: Shared Shopping Cart Example

Social commerce applications, such as eBay, cartonomy, flipkart, etc., are examples of such client applications for which enforcing application level correctness is important. Such applications are typically characterised by 1) very heavy concurrent workloads, and 2) multiple user groups with various different application roles. Depending on the user group for the user invoking the application, a different set of storage operations are invoked from a social commerce application. The correctness conditions for the Shopping cart (which is similar to cartonomy) and Bidding application (which has similarity to eBay.com), are given as: $bid_{amount} = \sum(price_{item}^{bid})$ and $if(role == admin) no_{items} = \sum(no_{items}^{user})$.

As a motivating example, a group of users from the same household or the same enterprise may use the same user account to purchase items through a shared online shopping

cart application, like Cartonomy: such shared accounts are typically used for generating a common bill for items purchased for a collaborative purpose. Shared carts are popular running examples for many research papers that study performance implications in replicated datastores, like the COPS system [19]. Multiple users may concurrently access the shopping cart, adding/deleting items to/from the cart at the same time (refer to the pseudocode in Figure 1). The users may also concurrently view the detailed invoice generated against a purchase. Under weak consistency settings and in the presence of network partitions, a user may observe that an invoice generated against a particular purchase contains items that are not currently present in his cart. Such anomalies can be expressed in terms of the variables that are used to denote the above quantities in the application code. Consistify allows the client application to explicitly specify the following correctness condition C : once an invoice has been generated, the cart viewed from a client application must contain exactly the same items that are present in the invoice. Let n_i^c and n_i^{inv} denote the number of items of type i under consideration in the cart and the invoice, respectively; $types$ denotes total number of possible item types. Using the arithmetic operator for summation, $\sum$, we quantify the number of items in the cart with the expression $\sum_{i=1}^{types} n_i^c$. Then, C_{n_i} can be given as: $\sum_{i=1}^{types} n_i^c = \sum_{i=1}^{types} n_i^{inv}$. In [28], we discuss the social networking application at Facebook, used by Lu et al. [20] as a running example. In this fashion, any such constraints comprising application level variables can be specified without resorting to any special-purpose language. The above client applications are representative of write-heavy workloads.

Figure 2: Architecture of Consistify

3 DESIGN OF CONSISTIFY

Inspired by the decoupled design principle followed by several state-of-the-art research prototypes [8], which separates API calls to the underlying datastore from the application-level business logic in the client application, Consistify is

designed as an interface layer (Figure 2) that upgrades an underlying geo-replicated datastore [11, 14, 16, 22] to improve application throughput by applying weak consistency settings, while respecting application specific correctness conditions and SLA deadlines. The above design cleanly separates the task of ensuring application specific concerns, like correctness and SLA deadlines, from the underlying datastore. renders flexibility to the interface, making it possible to be implemented on top of any geo-replicated datastore. To make the interface interact with a different datastore, the database connector and the accessor methods in the Execution Manager module (Figure 2) of the interface need to be modified. Also, this allows client-centric performance of the underlying datastore to be tuned according to application-specific requirements without modifying the code of the datastore.

The interface layer comprises multiple *interface instances*, one corresponding to each client. Each client application *Cl* uses a separate interface instance. Each interface instance is physically colocated with the respective client machine, i.e., the machine hosting the client application. *There is no inter-node communication between the interface instances.* An interface instance is implemented as a library module that provides a high level API to the client *Cl*, comprising methods to execute queries on the underlying datastore. The API does not depend on the developer's expertise of the underlying datastore, i.e., it does not require the developer to possess knowledge of the internal mechanisms of the underlying datastore. The API acts as an abstraction layer that allows application developers to develop client applications, which can respect correctness conditions and meet SLA deadlines, while executing queries on geo-replicated datastores. As discussed in Section 2, the correctness condition $\mathcal{C}$ is a postcondition that imposes constraints on the values of certain keys accessed by some of the queries invoked by *Cl*. $\mathcal{C}$ is supplied by the client *Cl* as an argument to the API methods. For each query in the client application, the developer calls a particular method from the Consistify API passing as arguments the correctness condition $\mathcal{C}$, the SLA deadline, and the values of each key.

The *dependency checker* module (Figure 2) analyzes the code of the client application *Cl* to extract the dependency relations among the queries in *Cl* in the form of Use Define chains (UD chains) [1]. The queries that do not affect $\mathcal{C}$ are reordered before the rest of the queries, and executed concurrently with the weakest consistency level (ONE for Cassandra), bypassing the rest of the steps mentioned below. The queries that affect the specified correctness condition, are buffered in an operation queue to control their execution order for preserving the correctness condition $\mathcal{C}$. The *consistency tuner* module (Section 3.2) computes a weak consistency level for each query, that respects the correctness condition $\mathcal{C}$ and the given SLA deadline. The Scheduler module (refer to the github version) determines an “optimal” execution order for the buffered queries, allowing each query to execute with a weak consistency level while not violating the

```

q1: val a = read(x); //reads the value at key x
q2: val b = read(y); //reads the value at key y
q3: sum = a+b; //adds x and y and store in sum
q4: write(z,sum); //writes sum to the key z
.....//some more application code
q5: c = read(z); //reads the value at key z

```

Figure 3: Example Query Sequence

```

DEF(x, q1), USES(x, q1), DEF(y, q2), USES(y, q2) = ∅;
DEF(sum, q3) = {q3}; USES(sum, q3) = {q4};
DEF(z, q4) = {q4}; USES(z, q4) = {q5};
DEF(z, q5) = {q4}; USES(z, q5) = ∅;

```

Figure 4: Use Define Chain for Above Example

correctness condition. Finally, the Execution Manager module executes the queries on the underlying datastore with the consistency levels chosen by the consistency tuner module.

The Failure Handling module waits for response from the datastore for each request until a pre-defined timeout period after which the operation is declared to have failed and marked for a retry. The module listens for error messages thrown by the underlying datastore, and retries failed operations, if the additional delay due to the retry does not risk causing violation of the SLA deadline. In case of violation of correctness condition or SLA deadline, it allows the user to rollback the datastore updates and marks the application as in violation of the SLA. During retries and SLA violations, the developers must take responsibility of freeing the control locks acquired during the failed execution. For resolving application execution errors, datastores typically throw error messages, and depend on the application developer to take appropriate corrective action. The developer must include failure handling code in the client application for handling possible query failures from the application-end. Developers must be able to 1) identify errors by error messages, and 2) deduce the causes of the errors. Consistify relieves developers from the task of handling all possible failures explicitly from the application code.

3.1 Dependency Checker Module

The non-relational models supported by geo-replicated datastores makes application of traditional static program analysis techniques [18] for dependency analysis unsuitable; hence we use a different approach here. Consistify uses UD chains for dependency analysis (Figure 2). For the running example in Figure 3, the query q_4 defines the key z (i.e., writes to z) that is used by q_5 . Similarly, q_4 uses the key sum that is defined by q_3 . Hence, $DEF(z, q_5) = \{q_4\}$ and $USES(sum, q_3) = \{q_4\}$ (refer to Figure 4).

We extract only one UD chain from each client application. The UD chains are used to determine the queries that affect the keys involved in the correctness condition, and segregate the queries in *Cl* into groups G^1 and G . We construct the groups G^1 and G using UD chains [28] as follows. Applying

static analysis on the code for a given client application Cl , we can determine which keys in the datastore are accessed by the queries comprising Cl . The above approach works under the assumption that the keys accessed by a query remain unchanged once the application starts execution. This assumption holds since the target applications comprise static sequence of queries, which in turn, implies that the group of queries are also static (fixed). Let K_{Cl} be the group of keys that are accessed by the queries invoked from Cl . Let K_C be the group of keys involved in the correctness condition, C , i.e., K_C is the group of keys that are present in the expression for C . The consistency tuner determines the group of keys, K , that belongs to both the groups K_{Cl} and K_C . Thus, $K = (K_{Cl} \cap K_C)$. It then determines the group of queries, G^I , such that: G^I comprises at least one write/update query, that can modify the value of at least one key in the group K . We form a group of queries G^I as follows. We search the extracted UD chains to identify queries that access at least one key from the group K , and add them to G^I . Thus, $G^I = \{q_i \in Q_{Cl} \mid q_i \text{ changes the value of at least one key } k_i \in K\}$. In other words, G^I comprises at least one write/update query, q_i , which affects the keys involved in the correctness condition C . Subsequently, we construct the group G from the remaining queries in Cl . Thus, $G = Q_{Cl} \setminus G^I$.

Consider that the running example of Figure 3 has an application-level latency threshold L_{SLA} . The datastore is initialized as follows: a value 1 stored at the key x , 2 at y , and 4 at z . The business logic for the above application requires the observed output (i.e., the value returned by the last query q_5) to be 3. Thus, the correctness condition for the above application is given by the logical predicate $z = 3$. The UD chains for the given application, extracted by the dependency checker, are given in Figure 4. For the example in Figure 3, the consistency tuner then determines the group of queries G , i.e., $G = \{q_1, q_2\}$. From the extracted UD chain and the given correctness condition, the dependency checker also determines that the queries q_3, q_4 , and q_5 belong to the group G^I , i.e., they require a stronger consistency guarantee, namely sequential consistency [28].

In their paper on RedBlue operations [18], Li et al. apply static analysis on a given application code, comprising a sequence of storage operations, to classify each operation as a red operation that requires stronger consistency settings, or a blue operation that requires weaker consistency settings, respectively; the static analysis approach assumes that the sequence of queries does not dynamically change. Similarly, the consistency tuner module of Consistify segregates the queries in Cl into groups G^I and G , based on the condition whether the queries affect the correctness condition C or not. Li et al. implement RedBlue consistency on a prototype system called Gemini that is based on a relational database, and do not consider SLA deadlines. On the other hand, Consistify extends an existing geo-replicated datastore to provide guarantees of meeting SLA deadlines and respecting correctness conditions through an interface layer. The interface layer does not use any datastore specific design parameters; and hence it can be modified to integrate

with any geo-replicated datastore. In addition, the RedBlue approach cannot ensure that the correctness conditions for an application are respected. This is because concurrent operations can modify the value of a key. To prevent such violation, Consistify provides a mechanism for “locking” the affected keys using control locks, and maintains a buffer (operation queue) for the operations that may affect the correctness condition. Consistify maintains control locks (lock variables) associated with each “affected key” from the group K_C , which are involved in a correctness condition; control locks are special-purpose key-value pairs collocated with its associated affected key in the datastore. Whenever a query executes on a key in K_C , it must first request to lock that key by writing a value “ON” to the control lock associated with that affected key (see details in [28]). Any other query which needs to access the same affected key as above can do so only if the value of the value of the control lock associated with that affected key is “OFF”. Consistify ensures that correctness conditions are respected by maintaining an operation queue for G^I queries (details in [28]), controlling the execution order, and maintaining the above system of locking on the affected keys.

3.2 Consistency Tuner Module

As explained, the queries in G^I affect the group of keys involved K_C in the correctness condition C . Hence, preserving C would require that clients always observe the correct values for the keys accessed by the queries in G^I . If a query q_i belongs to G^I , q_i affects the keys in the group K_C . Hence, the queries in Cl , that follow q_i in the invocation order of Cl , and access one or more keys updated by q_i , must observe the latest updates by q_i . For this to happen, the queries in G^I must be performed with stronger consistency guarantees as follows. Sequential Consistency (SC) [10, 24] requires that a global execution comprising operations executed from one or more clients must be equivalent to the result of executing the operations in a sequential order, such that the order among operations from each client in that sequence matches the invocation order of the operations. The consistency tuner module applies consistency settings, like sequential or causal consistency, for queries in G^I . Stronger consistency settings require coordination among a larger number of replicas, which in turn, can cause larger observed latency. The queries in G cannot affect the value of the keys involved in C . Hence, these queries can be concurrently executed under weaker consistency guarantee as follows. *Eventual consistency* is one of the weakest notions of consistency; it guarantees that in absence of any new updates to a given object, eventually an operation on that object will return the latest result [10]. Weak consistency levels, like ANY in Cassandra, can produce lower latency, since they require fewer of replica nodes to acknowledge to the coordinator before returning the result.

Concurrent execution of the queries implies that the latency for these queries do not compound, i.e., add up, towards the overall latency for the application. Since queries

in the group G execute in parallel with queries in group G^1 , T is computed as the maximum of the estimated aggregate completion times of the queries in group G and G^1 . Let the aggregate latencies of queries in G and G^1 be denoted by M and S , respectively. Then, T is the maximum of aggregate latencies M and S . We present details of latency estimation in [28], and discuss the computation of S below. Since the queries in G execute in parallel among themselves, M is computed as the maximum of the estimated latencies of queries in G . Assuming that Cl passes the *Schedulability Analysis*¹ [28], the *estimated completion time* T of the application is bounded by the SLA deadline, i.e., $T \leq L_{SLA}$, where L_{SLA} is the SLA deadline. For interactive applications the above SLA deadline is provided by the user as an input parameter to the client application either from the console or in the form of a string comprised in an input file. This, in turn, implies that L_{SLA} exceeds or equals the maximum of M and S . Since M is the maximum latency for queries in G with the weakest consistency level (ANY in Cassandra), the SLA deadline exceeds or equals M . Hence, no query in G can violate the SLA deadline, or negatively affect the observed throughput, under the weakest consistency level. Thus, the consistency tuner has to deliberate about the choice of consistency level only for the group of queries, G^1 , which actually affect the correctness condition. All queries in G are simply executed in parallel with the weakest consistency level.

Consistify does not address Byzantine failures; it only handles failures for queries in the group G^1 , since only they affect the correctness condition. The queries in the group G are allowed to fail, and failure messages for these queries are passed on to the developer; the developer must handle these failure at the application-level. For queries in the group G^1 , it stores the latest values of the keys involved in temporary key-value pairs. If there is a failure in the middle of writing the temporary key-value pairs, the above key-values are removed by the interface on the completion of a timeout period. If a query fails, the failure handler first removes the corresponding entry from the registry. Then it rolls back any updates made by that query on the datastore, and restores the old value of the keys involved in the failed query from the temporary key-value pairs. If there is a crash during the rollback itself, the interface again retries the process of rollback. Finally, the query is re-inserted into operation queue, where it waits to be picked by the Scheduler for execution. However, by enforcing retry of failed queries, the SLA deadlines may be violated in certain cases. If the above query failure causes the application to potentially violate the correctness condition, the updates due to the G^1 queries are rolled back. The affected keys are restored to their old values. Consistify retries the application by re-initializing the operation queue and the registry, and a message is sent to the user informing that the application is being retried. If the SLA is violated, the interface sends a failure message, along with the delay

by which the SLA was exceeded. It asks the user if the results can be accepted despite the SLA violation, or if the application should be retried. In case of the latter choice, it rolls back all updates, and re-inserts all the queries in the operation queue.

Consider a geo-replicated datastore [16, 22], configured with a replication factor R . Consider that queries q_i and q_j performed under consistency levels C_i and C_j , respectively, on a geo-replicated datastore [16, 22], configured with a replication factor R . Let the number of nodes that must respond with acknowledgement to the coordinator node, for queries q_i and q_j , be n_i and n_j , respectively. The values of n_i and n_j may vary from 1 to R for C_i and C_j varying from ONE to ALL, respectively, as in Cassandra [16]. The conditions for the pair of queries q_i and q_j to execute under causal and sequential consistency guarantee are: **Condition 1:** By definition, the number of replica nodes that can acknowledge to a query cannot exceed the total number of replicas R , i.e., $1 \leq n_i \leq R$ and $1 \leq n_j \leq R$ [16]. **Condition 2:** If a write query q_j defines a key k which is used by a read query q_i , then the consistency levels C_i and C_j must be such that the replicas responding read and write have one or more common nodes among them, i.e., $n_i + n_j > R$.

Condition 2 implies that the correct value gets read, the read and write quorums, i.e., the group of replicas acknowledging to the coordinator for read and write, respectively, must overlap. As an example of Condition 2, let us consider the case that q_j does not use any key defined by a prior query. Hence it can be run with the weakest possible consistency setting, which is ONE or ANY for Cassandra. We chose ONE here (we could have chosen ANY as well with the same effect), and apply $C_j = \text{ONE}$, which, in turn, implies $n_j = 1$. Hence, condition 2 dictates that n_i must be such that $n_i > R - 1$. But, condition 1 requires that $n_i \leq R$. Hence, if $R = 3$, the weakest possible consistency level for q_i is ALL, i.e., $C_i = \text{ALL}$ [16].

For the example in Figure 3, the queries in group G are directly passed on to the Execution Manager with ANY/ONE read-write consistency levels, bypassing the operation queue. Execution Manager can execute a query only if there does not exist an entry in the registry, comprising a conflicting query (from another client) that is currently accessing the same keys. Once any existing conflicts for the queries q_1 and q_2 have been cleared, they are executed over the datastore. Queries from group G^1 , namely q_3 , q_4 , and q_5 are inserted into the operation queue. The scheduler writes an entry for q_3 into the registry. The current deadlines for the query q_3 and q_4 , updated by the Scheduler, are denoted by L_{q_3} and L_{q_4} , respectively. For concurrent queries requesting given lock, the scheduler checks the registry entries to determine the current waiting time T^W for each query. It also estimates the aggregate latency L^{dep} of the queries that are dependent on a given query. The scheduler then determines which query has the least amount of time left to satisfy its current deadline L_{q_i} , where $i \in \{3, 4, 5\}$ [28]. It selects the query q_3 , which produces the minimum value for the expression $L_i - T^W - L^{dep}$. The scheduler requests a lock on the key

¹Schedulability Analysis determines if a given execution sequence will finish within a given deadline.

sum involved in q_3 . Once conflicts are cleared, the scheduler sets the *STATUS* column of the registry entry for q_3 to ON, and requests a lock on the key *sum*. Once q_3 acquires the lock, it calls the consistency tuner (refer to Section 3.2) to determine the weakest consistency level. The Execution Manager executes q_3 on the datastore with the consistency level predicted by the consistency tuner. Upon completion of execution, the Scheduler frees the lock for the key *sum* by setting the *STATUS* column to OFF, and removes the registry entry for q_3 . Once q_3 is removed from the registry, q_4 is released from the operation queue, and an entry is written to the registry for q_4 , and so on.

4 EVALUATION

Following state-of-the-art consistency tuning frameworks [29, 30], we run our experiments on a geo-replicated testbed comprising 3 c3.2xlarge Amazon EC2 instances, spread out across 2 regions (2 in us-west-2b, and 1 in eu-west), running Apache Cassandra 2.1.6 (one of the most widely used geo-replicated datastore) with a replication factor of 3, loaded on top of Ubuntu 16.10 LTS. State-of-the-art consistency tuning tools [9, 27] measure observed consistency in terms of observed *staleness*, i.e., how stale (old) a version of a data item is with respect to the latest update. However, we do not compute staleness, since it is not relevant to the problem statement; we are concerned with application-level correctness, not preventing stale results. Following Quela [29, 30], we perform experiments with three benchmark applications - stock trading, shopping cart, and online retail store. Banking applications are not ideally suited for applying weaker consistency settings since they work under strict correctness conditions that mandates stronger settings for geo-replicated datastores. The stock trading application comprises a sequence of operations identical to the bidding matrix application of Quela. The above client applications are invoked from remote c3.2xlarge Amazon EC2 instances running Ubuntu 16.10, which are not colocated with the datastore nodes.

4.1 Comparison With Fixed Consistency

Many storage systems, such as quorum-replicated systems built along the lines of the Dynamo system of Amazon [14], allow the developers to choose from a range of fixed consistency levels, which are tuning parameters that control the quorum of replicas that must acknowledge before returning result of an operation. It is a difficult task to choose the appropriate fixed consistency level that can allow the application to respect a given SLO deadline and the correctness condition. If the developer chooses a strong consistency level, like ALL, for all operations comprised in a given application, the high latency due to greater coordination may easily result in violation of the SLO deadline. An example of such a case is when a “non-admin” user adds or deletes items on/from a shared shopping cart (refer to Figure 1). To favour high availability, users are allowed to perform concurrent add/delete operations using weak consistency levels because individual users do not care about the number of

items added/deleted by other users in the shared carts as long as they are able to see their own items in the cart. On the other hand, if the developer only considers the SLO deadline, the developer may choose the weakest consistency level i.e., ONE, which produces lower latency but may result in higher chances of violation of correctness conditions. If the correctness condition imposes a weak constraint, and can be achieved with less coordination among replicas, the developer may apply such a weak consistency level to satisfy a stringent SLO deadline. On the other hand, if the correctness condition is highly restrictive and the SLO deadline is less stringent, then the developer must choose a strong consistency level, tolerating high latency. An example of such a case is when an admin user approves the items in a shopping cart and finishes the purchase order for a shared shopping cart (refer to Figure 1). Since the number of items in the shopping cart must be exact while submitting a finalized purchase order the developer must tune the shopping cart application with such a strong consistency level. In contrast, Consistify automatically tunes the storage system with the weakest consistency level per operation that does not cause violation of the correctness condition of the application. We use the number of operations per second to quantify average throughput (Figures 5(a), 5(c), and 5(e)), and 95 percentile latency, in milliseconds (Figure 5(b), 5(d), and 5(f)), obtained against varying number of clients, with consistency levels chosen with Consistify, and with manually chosen fixed consistency levels, namely ONE, QUORUM, and ALL, respectively. The results (Figures 5(a), 5(c), and 5(e)) indicate that Consistify succeeds in producing increased observed throughput over fixed manually chosen strong consistency settings except for weakest consistency level (ONE) since it never enforces coordination among replicas.

The mean throughput obtained with Consistify exceeds that with fixed strong consistency levels, namely QUORUM and ALL, by a margin of 10,000 operations/s on an average. On top of that, Consistify produces lower latency for 95% cases (Figures 5(b), 5(d), and 5(f)) than that with ALL and QUORUM. This is because without Consistify, all operations are performed with strong consistency setting, resulting in higher latency due to coordination among a greater number of replicas. But, Consistify applies one of the weakest possible consistency settings for each operation, choosing a particular setting according to a pre-defined arbitration rule (given in [28]) in the case of multiple weakest consistency settings. Further, without Consistify, operations are performed in a sequential blocking manner as per the program order. This results in an aggregate latency, which is a sum of the latencies of individual queries comprising the client application. On the other hand, Consistify controls the execution order of the queries, allowing the queries in the group G to be concurrently executed, resulting in lower latency. In this manner, Consistify results in an aggregate latency that equals the latency of the longest executing query in the client application. Also, the above technique improves the observed throughput, especially in comparison with stronger consistency levels (QUORUM and ALL). This can be attributed

Figure 5: Consistify vs QUELA vs Manually Chosen Fixed Consistency Level

to the increase in the number of concurrent operations with Consistify, since queries in group G would otherwise be executed sequentially without Consistify. Consistify matches the throughput values produced by the weakest consistency level (ONE) in 90% cases, with a maximum difference in throughput of 5,000 operations/s. This is because Consistify applies the weakest possible consistency settings for queries in group G , requiring lesser coordination among replica nodes, resulting in low latency.

We maintain a log file comprising results of the execution of the example applications where we mark whether the application achieved SLA or not, and measure the percentage of cases that resulted in correctness violation. Without Consistify, fixed consistency setting ONE is applied for all operations. On the other hand, Consistify applies stronger consistency settings for the queries in group G^1 , causing lower throughput. This also results in latency observed with Consistify to exceed the latency observed with fixed consistency ONE for 90% cases, but fixed consistency ONE results in correctness violation for more than 55% cases, whereas Consistify respects correctness conditions in 100% cases. Further, Consistify also meets the SLA deadline of 50 ms with a 97% success rate (Figures 5(b), 5(d), and 5(f)), whereas strong consistency levels, namely ALL and QUORUM, result in SLA violations for over 30% of cases; but they preserve correctness for over 75% cases. On the other hand, weakest consistency level (ONE) respects the SLA in 100% cases, but violates the correctness condition in more than 55% cases. The typical latency SLA for the Amazon shopping cart application is 300 ms [14], which is 6 times the

latency thresholds considered here; we use a more stringent bound to demonstrate the effectiveness of our interface in preserving such bounds.

4.2 Comparison with Quela:

Prior to Consistify, Quela was the only system which allows client applications to enforce correctness conditions on geo-replicated datastores. On top of that, Consistify allows applications to work under SLA deadlines, while Quela does not consider SLA deadlines. We compare the average throughput and 95 percentile latency obtained with Consistify, and that obtained with Quela, against varying number of clients (Figures 5(a), 5(c), 5(e), 5(b), 5(d), and 5(f)). The results with the benchmark applications demonstrate that Consistify outperform Quela, in terms of the obtained throughput, while also satisfying the SLA deadlines. Under identical conditions, we demonstrate that Consistify outperforms Quela, i.e., produces up to 10 times throughput. Consistify exceeds the throughput achieved by Quela in 95 % cases. The throughput with Consistify steadily increases with increasing number of clients. Contrastingly, the throughput with Quela remains almost constant throughout, this is because of the locking mechanism employed by Quela. Quela uses lightweight transactions that internally use Paxos messages. This adds to number of messages exchanged during each operation. The message passing overhead increases with the increase in the number of nodes, reducing scalability of Quela. Moreover, the throughput with Quela decreases as the number of clients increases beyond 3000. On top of that, unlike Quela, which does not consider SLA deadline at all,

Consistify achieves a 97 % success rate in meeting stringent SLA deadlines (50 ms), while also preserving the correctness conditions.

Also, Consistify produces lower latency than Quela, up to 150 ms lower (refer to Figures 5(b), 5(d), and 5(f)). This is because Consistify executes the queries in the group G with weak consistency guarantees in parallel with the queries in group G^I that require strong consistency guarantees. Unlike Consistify, Quela does not control the execution order of the queries in a given application; it only determines the weakest possible consistency level for each query. Consistify divides the queries comprising a client application Cl into two groups G^I and G , depending on whether the queries affect the correctness condition or not, respectively. Consistify executes the queries in G with weakest consistency level, namely ONE. Further, the queries in G are executed in parallel with the queries in G^I ; G^I queries are scheduled to execute by the Scheduler, which are executed in accordance to an appropriate schedule devised by the Scheduler module. This enables Consistify to reduce the overall observed latency of a client application, which in turn, results in drastic improvement in throughput (up to 10 times over Quela).

Controlling the execution order, Consistify executes the queries in the group G with weak consistency guarantees in parallel with the queries in group G^I that require strong consistency guarantees. Contrastingly, for providing strong consistency guarantees, Quela uses the lightweight transaction primitive of Cassandra [15], which increases the framework overhead due to multiple Paxos rounds; thus resulting in high latency. Contrastingly, Consistify executes only the queries in G^I in a blocking manner, using control locks on the keys involved in the correctness conditions. It executes operations in G in parallel with queries in G^I , bypassing the operation queue. Hence, compared to Consistify, Quela produces higher latency, and has greater chances of violating stringent SLA deadline, when executed under stronger consistency guarantees. In comparison to Consistify, Quela does not scale with increasing number of clients, because of performance bottlenecks resulting from reasons explained in [28].

4.3 Other State-of-the-art Frameworks:

Bailis et al. [8] provide causal consistency on top of datastores by snapshot merging techniques; they do not support consistency guarantees stronger than causal. The experiments in this paper were run on a cluster consisting of 5 servers each comprising 8 virtual CPU cores, 15 GB memory, 2*80 GB SSDs and 40 GB storage. Identically, Bailis et al. ran their experiments on 8 CPU cores (ECUs), and 15 GB memory; they use 4*420 SSDs, but SSDs do not directly contribute to computation speed in this case. With single node configurations, Bailis et al. achieve peak throughput of 10,000 to 1,000,000 operations/second. However, with 2-DC configurations, they produce peak throughput between 5,000 to 50,000 operations/second (Figure 6 in [8]). In 2-DC settings, Consistify beats [7], producing average throughput

(which is less than peak throughput) scaling from 5,000 to 50,000 operations/second with number of clients increasing from 1,000 to 6,000 (refer to Figures 5(a), 5(c), and 5(e)); while also satisfying correctness conditions and SLA deadlines. Though the average throughput is identical in both cases, Consistify produces higher peak throughput. SIEVE [18] applies static analysis for automated consistency tuning for relational databases. Quorum-replicated stores support simple, non-expressive datatypes, which are unsuitable for the static program analysis approach of Li et al. The experiments in SIEVE were run on a cluster of 2*6 CPU cores and 48 GB memory, which is a considerably higher powered testbed, Figure 9 in [18] shows that SIEVE produces only 4,000 requests/second with a 2-site (DC) configuration, whereas Consistify produces up to 50,000 operations/second with 6,000 clients with a 2-DC configuration.

Cure [2, 23] is a prototype distributed database that provides causal consistency on quorum-replicated datastores, the strongest possible consistency setting that can provide availability guarantees under partitions, using CRDTs [26]. Further, the solution approach followed by Cure is a new prototype datastore, hence they present observed throughput with increasing number of servers to demonstrate the scalability of the datastore. Contrastingly, Consistify provides an interface that can be “plugged in” to an existing datastore to provide correctness and latency guarantees with improved throughput. Hence, our approach can enable existing application to experience better user experience with minimal changes to application design, with the same underlying datastore. Because of this, with Consistify, the underlying datastore is a black box, and we model the performance of the framework with respect to its interaction with client application. Hence, unlike Cure, we are not concerned with how observed throughput changes against varying number of servers in the underlying datastore; rather we are interested in how the framework scales against increasing number of clients. The testbed for Cure comprises 2*8 CPU cores and 128 GB memory, which is more than doubly powerful than the testbed for Consistify. We compare our 2 DC, 3 nodes results with the results in [2] obtained by running Cure on a 2 DC 25 server configuration. The average throughput per server with Cure ranges from 14,000 to 22,000 operations/second per server against varying read proportions. On the other hand, with a geo-replicated configuration and a contrastingly smaller testbed, Consistify (across 2 regions) records an average throughput per server that steadily scales from 5,000 to 16,667 operations/second per server with 6000 clients.

5 CONCLUSIONS

Consistify provides an abstraction layer that allows application developers to tune client applications with weakest possible consistency settings that enable applications to satisfy a given SLA deadline and respect correctness conditions while executing storage operations on a geo-replicated datastore.

Consistify outperforms the state-of-the-art systems by a factor of 2 to 10, over geo-replicated datastores.

6 ACKNOWLEDGEMENT

We thank Dr. Doug Terry for helping us in developing the idea behind this work. The project is partially funded by Army Research Office (ARO) and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA.
- [2] D. D. Akkourath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Pregaia, and M. Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 405–414.
- [3] Peter Bailis. 2015. The Case for Invariant-Based Concurrency Control. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*.
- [4] Peter Bailis, Alan Fekete, Michael Franklin, Ali Ghodsi, Joseph Hellerstein, and Ion Stoica. [n. d.]. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *SIGMOD 2015*. New York, NY, USA, 1327–1342.
- [5] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196.
- [6] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable atomic visibility with RAMP transactions. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 27–38.
- [7] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. [n. d.]. Bolt-on causal consistency. In *SIGMOD 2013*. New York, NY, USA, 761–772.
- [8] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 761–772.
- [9] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow.* 5, 8 (2012), 776–787.
- [10] K. Birman and R. Friedman. 1996. *Trading Consistency for Availability in Distributed Systems*. Cornell University. Department of Computer Science.
- [11] Brad Calder, Ju Wang, Aaron Ogus, Niranjana Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 143–157.
- [12] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaiice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. 178–188.
- [13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 251–264.
- [14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220.
- [15] Sivaramakrishnan KC. 2015. Quela. <https://github.com/kayceesrk/Quela/blob/master/src/Quela/>.
- [16] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.
- [17] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691.
- [18] Cheng Li, João Leitão, Allen Clement, Nuno M. Pregaia, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. 281–292.
- [19] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. 401–416.
- [20] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 295–310.
- [21] Christopher Meiklejohn. [n. d.]. Riak PG: Distributed Process Groups on Dynamo-style Distributed Storage. In *Erlang 2013*. New York, NY, USA, 27–32.
- [22] Christopher Meiklejohn. 2013. Riak PG: distributed process groups on dynamo-style distributed storage. In *Proceedings of the Twelfth ACM SIGPLAN Erlang Workshop, Boston, Massachusetts, USA, September 28, 2013*. 27–32.
- [23] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. [n. d.]. The CISE tool: proving weakly-consistent applications correct. In *PaPoC@EuroSys 2016*. 2:1–2:3.
- [24] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653.
- [25] Sunita Sarawagi and Michael Stonebraker. [n. d.]. Reordering Query Execution in Tertiary Memory Databases. In *VLDB 1996*. 156–167.
- [26] Marc Shapiro, Nuno M. Pregaia, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. 386–400.
- [27] Subhajit Sidhanta, Wojciech M. Golab, Supratik Mukhopadhyay, and Saikat Basu. 2016. OptCon: An Adaptable SLA-Aware Consistency Tuning Framework for Quorum-Based Stores. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. 388–397.
- [28] Subhajit Sidhanta, Supratik Mukhopadhyay, and Wojciech Golab. 2016. *Consistify: A Correctness-preserving, SLA-Aware, Automated Consistency Tuning Framework*. Technical Report.
- [29] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jaganathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). 413–424.
- [30] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jaganathan. 2016. Representation without Taxation: A Uniform, Low-Overhead, and High-Level Interface to Eventually Consistent Key-Value Stores. *IEEE Data Eng. Bull.* 39, 1 (2016), 52–64.