Consistify: A Framework for Safe and Fair Execution of Concurrent SLA-driven Client Applications on Quorum-based Datastores

Subhajit Sidhanta[1], Supratik Mukhopadhyay[1] and Wojciech Golab[2]

[1] Louisiana State University
[2] University of Waterloo

Abstract. We investigate the problem of guaranteed preservation of application level correctness in quorum-based datastores under concurrent application executions, while satisfying latency thresholds specified in the SLA (Service Level Agreement). Consider a sequence of operations (or a single operation) on a quorum-based datastore invoked from a client application, with a application-level latency threshold specified in the SLA, and a correctness condition specified by the user as the post condition. We present Consistify, a novel framework that can enable execution of client applications on quorum-based datastores with the weakest possible consistency setting, while simultaneously satisfying the specified correctness conditions and the latency threshold, given in the SLA. As per our knowledge, Consistify is the first documented work that provides high level API support that allows application developers to develop client applications that can preserve the specified application level correctness conditions and given SLA deadline, while performing operations on quorum-based datastores. Consistify achieves the above goals without any dependency on the developer's knowledge of the mechanisms of the underlying datastore, without any additional declarative language for specifying the application level guarantees, and without any additional overload, in terms of system resources and processing delay. Using benchmark workloads, we experimentally demonstrate that Consistify outperforms state-of-the-art systems by a factor of 10, while also satisfying the given SLA deadlines.

1 Introduction

Many quorum-based distributed data stores, like Cassandra, Riak, and DocumentDB [14, 19, 7], allow users/developers to explicitly declare the desired client-centric consistency [3] setting for an operation. Such systems accept the client-centric consistency settings for an operation in the form of a runtime argument, typically referred to as the *consistency level*. According to consistency level applied, the system waits for coordination among a specific number of replicas

[3] *Client-centric consistency* deals with consistency from the viewpoint of the client application .

containing copies (i.e., versions) of the data item accessed by the given operation [14]. The latency for a given operation depends on the waiting time for the above coordination; hence, in turn, depends on the consistency level applied. For example, a weak consistency level for a read operation in Cassandra [14], like READ ONE, requires only one of the replicas to coordinate successfully, resulting in low latency and high chances of a stale read.

Quorum-based datastores [14, 19, 7] often tradeoff support for serializability and transactions (enforced by application of stronger consistency levels) in favor of availability and partition tolerance. Hence they can not simultaneously guarantee correctness of application execution, and at the same time, satisfy latency requirements, especially with concurrent executions [14, 19, 7]. In this context, by correctness we primarily mean the application level safety properties (liveness is handled internally by the datastore), that involve anomalies observed by the client due to conflicting updates by concurrent operations. The correctness conditions are application level invariants that are either specified by the users, or are implicitly defined in the application semantics. Such correctness conditions can not to be handled by the built-in mechanisms of quorum-based datastores alone, especially with weak/eventual consistency settings. Instead, quorum-based datastores provide users the choice of running applications with weak consistency guarantees to experience lower observed latency, increased availability, and partition tolerance [6, 11]), at the risk of undesired anomalies in the data, like negative balance in a bank account or duplicate values for an unique data item. We investigate the possibility of automating the task of determining the weakest consistency level that can preserve the correctness of the given operation, i.e., that can produce the correct output as per the semantics of the operation, while at the same time satisfy the SLA threshold for latency.

1.1 Contributions

We present Consistify, a novel framework that automatically tunes quorum-based datastores with the weakest possible consistency level, to execute a given client application under user-specified correctness conditions and given SLA deadlines. Consistify allows client applications to automatically tune the same underlying datastore to provide any required consistency guarantee (like serializability, causal consistency, eventual consistency, etc.). As per our knowledge, Consistify is the first documented work that allows application developers to develop client applications, that can preserve the application level correctness conditions while performing operations on quorum-based datastores, while also simultaneously satisfying the given SLA deadline. Consistify enables client applications to achieve the above goals through a high level API that does not require the developer's to have any knowledge of the mechanisms of the underlying datastore. While the state-of-the-art systems [21] requires knowledge of programming languages for specifying required application level guarantees, Consistify uses simple first order logic expressions to capture the latency SLA and the correctness condition. Consistify uses lightweight data structures to monitor the dependency relations in the given query sequence, minimizing the usage of local memory and disk. Contrastingly, the state-of-the-art systems either store snapshots or keep local copies of the data, or use the lightweight transaction primitive of Cassandra [14]. The former approach puts additional overhead on the memory and disk usage of the system, while the second approach is unreliable, and causes additional processing delay due to additional Paxos roundtrips [14]. Using benchmark workloads, we experimentally demonstrate that Consistify outperforms state-of-the-art systems (can produce 10 times the observed throughput than the state-of-the-art), while also satisfying the given SLA deadlines.

1.2 Correctness Conditions of Example Use Cases

Consistify accepts the correctness conditions for a client application from the users in the form of first order logic relations. Each client application comprises a sequence of *queries* (i.e., storage operations on the underlying datastore, following the standard terminology of storage systems), and the given correctness conditions impose constraints on the values returned by the queries. We illustrate our problem with two example use cases, namely stock trading, and shopping cart applications, and discuss a sample correctness condition for each case. The use case for a shopping cart application may comprise a sequence of queries that perform the operations: 1) browse the catalog, 2) update the cart with selected items, 3) sign in, 4) submit the order, 5) review the invoice, and 4) pay the invoice amount. The correctness condition for the shopping cart application may specify the following constraint regarding the final invoice amount billed for a purchase: $invoice_{amt} = \sum_{k=1}^{item_{types}} n_{item} \times price_{item}$ where n_{item} is the number of items of each type, $price_{item}$ is the unit price of each type of item, $item_{types}$ is the number of the different item types chosen, and $invoice_{amt}$ is the total invoice amount for the purchase. A stock trading application, used for buying/selling of stocks, may perform the following sequence of operations: 1) the investor logs in to the system, 2) the investor chooses buy or sell as the operation type, 3) the investor chooses the name and the number of stocks to trade, 4) system checks the deposit and stock availability for the transaction, 5) if everything is satisfied, system prompts for the order confirmation, 5) once the investor confirms the order, the order detail is recorded in the investor's order history. The correctness condition may state that: $\sum_{stocks} n_{stock} \times price_{stock}^{buy} \leq \sum_{stocks} n_{stock} \times price_{stock}^{selling}$ where n_{stock} are the number stocks being sold,$price_{stock}^{buy}$ is the unit buying price of a stock, and $price_{stock}^{selling}$ is the unit selling price of a stock, and $\leq$ is the arithmetic operator for the less than equals relation.

2 Design of Consistify

Consistify (Figure 1) comprises an interface layer that client applications use to interact with the underlying quorum-based datastore. The interface is implemented as a library module, collocated with the client application. The interface

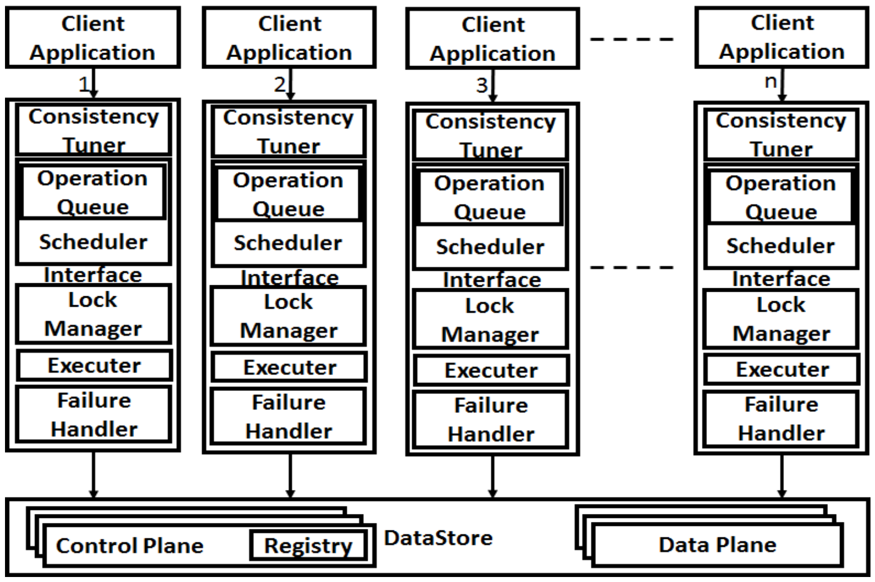


Fig. 1: Architecture of Consistify

provides an API comprising methods that can be called from client applications to execute storage operations on the underlying datastore. The above API acts as an abstraction layer that allows application developers to develop client applications that can execute on quordum-based datastores, under given correctness condition and SLA deadline, without knowledge of the mechanisms of the underlying datastore. For each query in the client application, the developer calls a particular method of the Consistify API, passing as arguments the given correctness condition, the SLA deadline, and the values for each key. The *dependency checker* module of the interface extracts the dependency relations in the form of an *UD chain* (i.e., an use-define chain) [2, 12] from the query sequence given in the client application.

The interface also comprises a *consistency tuner* module (Figure 1) that determines the weakest possible provably valid consistency level, under the given SLA and correctness conditions. The *operation queue* (Figure 1) maintains a buffer to control the execution order of the queries that require serialiablity/-causality for preserving the given correctness condition. In a quorum-based datastore, objects are stored using simple non-expressive data types, like key-value pairs, with support for only basic operations read and write operations (read the latest value for a key and write a value to a key). Consistify partitions the underlying datastore into a dataplane (Figure 1), that stores the actual dataset, and a control plane, that maintains a *registry* of control flags. In the case of requirement of serializability, the *scheduler* module checks if there exists a conflicting query that requires access to the same key, In case of existence of a conflicting query, the query registers its intent to access the key in question by writing a control flag entry $entry_k$ to the registry. $entry_k$ is a tuple $\langle k,\ q_{Cl_{id}i},\ Cl_{id},\ T^{W}_{Cl_{id}i},\ L^{dep}_{Cl_{id}i},\ L^{d}_{Cl_{id}i},\ STATUS\rangle$ comprising: 1) the name of the key k, 2) an unique id Cl_{id}, identifying the identity of the client application invoking the query $q_{Cl_{id}i}$ in question, 3) the signature of the query statement $q_{Cl_{id}i}$, 4) the waiting time of $T^{W}_{Cl_{id}i}$ of the query, initialized with the timestamp difference T - $T^{inv}_{Cl_{id}i}$, where $T^{inv}_{Cl_{id}i}$ is the timestamp of the invocation of the query, and T is the current timestamp, 5) $L^{dep}_{Cl_{id}i}$, which is the sum of the estimated latency of all the queries dependent on $q_{Cl_{id}i}$, given as $\sum_{i=1}^{n} L^{Est}_{Cl_{idi}}$, where n = the number of queries dependent on $q_{Cl_{id}i}$, 6) the current deadline $L^{d}_{Cl_{id}i}$ of the client application, initialized with the latency threshold L_{SLA} specified in the SLA for the given application, and 7) a flag $STATUS$ that represents whether the query has already been scheduled, i.e., takes the values ON or OFF. From the time of invocation till the completion of execution of the client application, the timestamps for the waiting time and the current deadline in $entry_k$ are regularly updated as: $T^{W}_{Cl_{id}i} = T$ - $T^{W}_{Cl_{id}i}$, and $L^{d}_{Cl_{id}} = L^{d}_{Cl_{id}}$ - (T - $T^{inv}_{Cl_{id}}$), respectively. The scheduler module devises the schedule, i.e., determines the execution sequence of the queries waiting in the registry, based on the values of the entries $entry_k$. Once the conflict has been cleared, the control is passed on to the *lock manager* module. It enables the query to acquire a lock on the concerned key by setting the value of the lock column in the datastore to an unique id, identifying the given query and the client application. Algorithm 2 is the algorithm for our lock manager module implementation.

2.1 Consistency Tuner: Determining Provably Valid Consistency Settings

For the given quorum-based store [14, 19, 7], consider that *Rep* is the given replication factor. For a given query $q_{Cl_{id}i}$ performed on the store with a consistency level given by the nominal attribute $C_{Cl_{id}i}$, $frac \times 1/Rep$ number of nodes must respond with acknowledgement, where the parameter *frac* varies from 1 for consistency level $C_{Cl_{id}i}$ = ALL to *Rep* for $C_{Cl_{id}i}$ = ONE, in Cassandra. The queries that do not use or define (see the use define chain section in the extended version at `https://github.com/ssidhanta/Consistify`) the keys involved in the correctness condition, can be concurrently executed with eventual consistency settings, i.e., with the weakest consistency level (like ANY/ONE in Cassandra). Such queries execute under eventual consistency guarantee, and they execute concurrently with other queries. Hence, these queries don't affect the SLA deadline. Hence, the consistency tuner considers the SLA deadline only for queries that require causality/serializablilty guarantees. Consider that $frac_i \times 1/Rep$ and $frac_j \times 1/Rep$ are the number of nodes that must respond corresponding to the given consistency levels $C_{q_{Cl_{id}i}}$ and $C_{q_{Cl_{id}j}}$ for queries $q_{Cl_{id}i}$ and $q_{Cl_{id}j}$, respectively, where $1 \leq frac_i \leq Rep$ and $1 \leq frac_j \leq Rep$ [14, 19, 7]. The condition for the pair of queries $q_{Cl_{id}i}$ and $q_{Cl_{id}j}$ to execute under causal guarantee demands the following: If a write query $q_{Cl_{id}j}$ defines a variable *var* that is used by the read query $q_{Cl_{id}i}$, then consistency levels $C^{mat}_{q_{Cl_{id}i}}$ and $C^{mat}_{q_{Cl_{id}j}}$ must be such that

$frac_i \times 1/Rep + frac_j \times 1/Rep \geq Rep$. The above rule is the minimal necessary condition for reading correct writes in quorum-based stores [14, 19, 7]. The above condition ensures that the quorum of replicas accessed for the read and write overlaps, thus guaranteeing that the correct value gets read. For example, $C^{mat}_{q_{Cl_{id}j}}$ = ONE implies $frac_j \times 1/Rep = 1$. Hence, for the above condition to hold, it requires that $frac_i \times 1/Rep \geq Rep - 1$, which in turn requires that $frac_i \geq 1$. The only consistency level for which $frac_i \geq 1$ holds is ALL, hence $C^{mat}_{q_{Cl_{id}i}}$ must be ALL for satisfying the condition U.

3 Use Define Chains

We refer to a given query to have *defined* a given variable (i.e., key), if it is the last write query that accessed this variable prior to the given query. Similarly a given query is said to *use* a given variable, if: 1) in the case of a read query, it is the first query to have read the concerned variable defined in a prior query 2) in the case of a write query, it is the first query to update another variable with the value of the concerned variable defined by a prior query. Use define chains (UD) [2, ?] can be used to determine: 1) the query that uses ($USES$) a particular variable defined by a given query, and 2) the query which defines (DEF) a variable that is being used by the given query. Consistify applies UD chains to identify dependency among the individual queries in a given query sequence. UD chains are based on functions DEF and $USES$ which are defined as: if the variable var used in a given query $q_{Cl_{id}i}$ is defined by a prior query $q_{Cl_{id}j}$ invoked from a client application Cl_{id}, $DEF(var, q_{Cl_{id}i}) = q_{Cl_{id}j}$, and $USES(var, q_{Cl_{id}j}) = q_{Cl_{id}i}$. The above dependency is used to derive the safety/correctness properties that need to be satisfied for safely executing a given query on the system. The UD chains for the given query sequence are developed as follows.

```
DEF(x, $q_{i1}$) = $\phi$; USES(x, $q_{i1}$) = $\phi$;
DEF(y, $q_{i2}$) = $\phi$; USES(y, $q_{i2}$) = $\phi$;
DEF(sum, $q_{i3}$) = $q_{i3}$; USES(sum, $q_{i3}$) = $q_{i4}$;
DEF(z, $q_{i4}$) = $q_{i4}$; USES(z, $q_{i4}$) = $q_{i5}$;
DEF(z, $q_{i5}$) = $q_{i4}$; USES(z, $q_{i5}$) = $\phi$;
```

3.1 The Consistify Scheduler: Controlling the Execution Order

Firstly, the scheduler performs schedulability analysis for the query sequence comprising the given client application under the given SLA, and the correctness condition. The queries requiring eventual consistency can execute concurrently with the weakest consistency level, producing lowest latency. Hence, the aggregate completion time of these queries is estimated as the latency of the query with the maximum estimated latency. On the other hand, the overall completion time for queries requiring causality/serializability is approximately estimated as the sum of the estimated latencies of the queries. Hence, the given application is schedulable if the SLA equals or exceeds the maximum of the following estimated times: 1) the maximum of the estimated latencies of the queries requiring

Algorithm 1 The Consistify Fair Scheduler

```
procedure SCHEDULER($registryState$)    ▷ the Fair Scheduler which schedules a query from the registry to acquire a lock on the given query
    returns void
    Let the state of the registry $registryState$ be given by the current entries $entry_i$ in the registry for each query $q_{Cl_{id}i}$ invoked by each client $Cl_{id}$ be given as $\langle k,\ q_{Cl_{id}i},\ Cl_{id},\ T^{W}_{Cl_{id}i},\ L^{dep}_{Cl_{id}i},\ L_{Cl_{id}}i^{d}, STATUS\rangle$
    $j = 0,\ min = 0;$
    if $checkSchedulability \leftarrow true$ then
        while $j \leq size(registryState)$ do
            $\langle k,\ q_{Cl_{id}i},\ Cl_{id},\ T^{W}_{Cl_{id}i},\ L^{dep}_{Cl_{id}i},\ L_{Cl_{id}}i^{d}, STATUS\rangle = \mathrm{read}(entry_j);$
            if $STATUS \not\leftarrow ON \wedge min \geq L_{Cl_{id}}i^{d} - T^{W}_{Cl_{id}i} - L^{dep}_{Cl_{id}i}$ then
                $min\ =\ L_{Cl_{id}}i^{d} - T^{W}_{Cl_{id}i} - L^{dep}_{Cl_{id}i}$
                $x = Cl_{id};$
                $q = q_{Cl_{id}i}$
                $write(entry_j, \langle k,\ q_{Cl_{id}i},\ Cl_{id},\ T^{W}_{Cl_{id}i},\ L^{dep}_{Cl_{id}i},\ L_{Cl_{id}}i^{d}, ON\rangle);$
            else
                j++;
            end if
        end while
        requestLock(x,k,q,timeOut);    ▷ Call the module requestLock to request a lock on the key $k$ with a preconfigured $timeOut$
    end if
end procedure
```

eventual consistency, and 2) the sum of the estimated latency for the queries requiring serializability or causality. If the application is schedulable, the scheduler first buffers the concurrent queries that have registered their intent to execute on the datastore by making entries in the registry. For concurrent queries from multiple clients trying to access a given lock, the scheduler checks the registry entries to estimate the following quantities: 1) the waiting time $T^{W}_{Cl_{id}i}$ for each query, and 2) the aggregate latency $L^{dep}_{Cl_{id}i}$ for the queries that are dependent on a given query. The scheduler then determines which client has the least amount of time left to satisfy its current deadline $L_{Cl_{id}}i^{d}$. Thus, it selects the query $q_{Cl_{id}i}$ which produce the minimum value for the expression $L_{Cl_{id}}i^{d} - T^{W}_{Cl_{id}i} - L^{dep}_{Cl_{id}i}$, and marks $q_{Cl_{id}i}$ by setting the value of $STATUS$ column in the registry to ON. The selected query then requests to acquire the lock on the concerned key. When the lock is free, the query acquires the lock by setting the lock column to its unique client id. Upon completion of the query execution, the lock is freed, and the corresponding entries in the registry and the operation queue are removed. The scheduler is fair, i.e., it provides each query in the registry the chance for serialized execution by acquiring locks on the datastore in a fair manner, where no query is starved indefinitely beyond a predefined threshold.

Algorithm 2 The requestLock Module

```
procedure REQUESTLOCK($x$, $q$, $k$, $timeOut$) ▷ Requests to acquire a lock on the key $k$
    Inputs: String $x$: the unique id of the client that requests the lock,
String $q$: the query that requests the lock,
String $k$: the key on which the query requests the lock,
Long $timeOut$ denoting the timeout for the lock request.
    returns void
    Let the state of the registry $registryState$ be given by the current entries $entry_i$ in the registry for each query $q_{Cl_{id}i}$ invoked by each client $Cl_{id}$ be given as $\langle k,\ q_{Cl_{id}i},\ Cl_{id},\ T^{W}_{Cl_{id}i},\ L^{dep}_{Cl_{id}i},\ L_{Cl_{id}}i^{d}\rangle$
    Long $lockState = -999999$, $timestamp = currentTime$, $Booleansuccess = false$
    while $timestamp \leq timeOut$ do
        $lockState = read(lock(k))$      ▷ read the value at column lock for the key $k$
        if $lockState \leftarrow free$ then
            $lock(k) = x$
            $ret = execute(q)$
            if $ret == true$ then
                $removeEntry(x, q, k)$      ▷ Call the module $removeEntry$ to remove the respective registry entry
                $removeQueueEntry(x, q, k)$ ▷ Call the module $removeQueueEntry$ to remove the respective entry from the operation queue

            end if
        else
            wait;
        end if
    end while
    if $success \leftarrow false$ then
        $resetRegistryEntry(x, q, k)$      ▷ Call the module $resetRegistryEntry$ to mark the query as filed and setting it for a retry in next scheduling cycle
        $lock(k) = free$
    end if
end procedure
```

3.2 Failure Handling

State-of-the art quorum-based stores [14, 19, 7] return error messages to the console during failure of execution of queries, but depends on the developers to take action with respect to each kind of failure. Consistify relieves the developers from the task of explicitly handling failures, which requires the developers to have the ability to identify the various error messages (and the causes of the errors) just from their names. The Consistify failure handler collects and interprets the error messages, and ensures correctness of the application despite such failure, while relaxing on the SLA deadline in extreme cases. The failure handler only handle failures for queries that require causality or serializability guarantees, since by design (Section 2) only those queries affect the correctness condition. It stores the last value of the key updated by the query in a temporary variable. If the query fails with a replica unavailability exception or a read timeout (Consistify does not addresses Byzantine failures), the failure handler removes the corresponding entry from the registry, rolls back any update made by that query, restores the old value of the respective key from the temporary variable. The query is then re-inserted into operation queue, and waits for execution. However, by enforcing retry of queries on failures, the SLA deadlines may be violated in certain cases at the cost of correctness of execution. If the application fails to satisfy the correctness condition, the updates due to the serializable queries are rolled back, the concerned keys are restored to their old values, 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 overshoot. It also asks the user if the user accepts the results despite the SLA violation, or if the application should be retried. In case of the latter choice, the interface rollbacks the updates, and re-inserts the queries in the operation queue.

4 Estimating the Latency for an Applied Consistency Level

The verifier module of Consistify determines the weakest possible consistency level to satisfy the latency threshold of the SLA. To perform this task, Consistify must estimate the latency for a given query performed with each possible consistency level on the given datastore. The latency observed for a given query performed with a given consistency level varies with respect to the replication factor of the datastore, the network conditions of the cluster. In the absence of a mathematical relation binding the applied consistency level to latency, the latency must be estimated either with training or approximated with benchmark workloads. The approach of using training requires large training data and has the pitfall of overfitting. Instead, Consistify uses 95 percentile latency estimates obtained by running YCSB [8] benchmark workloads with each possible consistency level on clusters comprising same number of nodes and replication factor.

5 Putting the Pieces Together: How Consistify Works

We illustrate how Consistify handles the given problem of consistency tuning on a quorum-based datastore with the running example in Figure 2, comprising a sequence of queries q_1 to q_5 invoked from a client application Cl_{id}, and a latency threshold SLA for the application specified in the SLA. The datastore is initialized as follows: the value 1 stored at the key x, 2 at y, and 4 at z. Consider that the business logic for the above application code require that the observed output, i.e., the value returned by the query q_5, is 3. Thus, the correctness

```
q1: read(x); //reads the value at x
q2: read(y); //reads the value at y
q3: sum=x+y; //adds the values x and y and stores in temporary
     variable sum
q4: write(z,sum); //writes sum to the key z
q5: read(z); //reads the value at z
```

Fig. 2: Example Query Sequence

condition for the above application is that the observed output must be 3, i.e., $z = 3$. We determine the weakest possible consistency level that can be applied for each query in the above query sequence, and the required execution order of the given query sequence, such that the above correctness condition is preserved. We use the notation $q_{Cl_{id}i}$ to denote the invocation of a query q_i from a client Cl_{id}.

Once invoked, the client application calls the dependency checker module of Consistify (Figure 1), which is collocated with the client. The dependency checker extracts the dependency relations among the queries in the client application in the form of an UD chain. The UD chains for the given query sequence are developed as follows.

```
DEF(x, $q_{i1}$) = $\phi$; USES(x, $q_{i1}$) = $\phi$;
DEF(y, $q_{i2}$) = $\phi$; USES(y, $q_{i2}$) = $\phi$;
DEF(sum, $q_{i3}$) = $q_{i3}$; USES(sum, $q_{i3}$) = $q_{i4}$;
DEF(z, $q_{i4}$) = $q_{i4}$; USES(z, $q_{i4}$) = $q_{i5}$;
DEF(z, $q_{i5}$) = $q_{i4}$; USES(z, $q_{i5}$) = $\phi$;
```

The interface performs schedulability analysis (Section 3.1) for the given client application under the given SLA, and the correctness condition. Then, it determines the queries that do not access the keys referred to in the correctness conditions, i.e., q_1, and q_2. Eventual consistency is sufficient for these queries; hence they are directly passed on to the verifier, bypassing the operation queue. The verifier determines that ANY/ONE read-write consistency levels are the weakest possible valid consistency levels for q_1, and q_2 under the given SLA. A given query can execute 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 the conflict for each of the queries q_1 and q_2 has been cleared, they are concurrently executed over the datastore, with the consistency levels predicted by the verifier.

From the UD chain and the given correctness condition, the dependency checker determines that the queries q_3, q_4, and q_5 require pairwise serializability. Hence, q_3, q_4, and q_5 are inserted into the operation queue (Figure 1). The scheduler (Figure 1) first writes an entry for q_3. At an instant of time, determined by the scheduling algorithm [4], q_3 is chosen among the registry entries.

[4] The scheduler (refer algorithm 1) regularly updates the deadlines of the queries in the registry with respect to the SLA deadline, the invocation time, and the current

The scheduler sets the *STATUS* column of the registry to ON, and calls the requestLock algorithm for requesting a lock on *sum*. setting the lock column to an unique id, identifying q_3 and the client application.

Once q_3 acquires the lock, it calls the verifier (Figure 1) to determine the weakest possible consistency level. The verifier uses the weakest precondition $P_{q_{Cl_{id}}3}$ (Section 2.1) to determine the set of weakest matching consistency levels $C^{mat}_{q_{Cl_{id}}3}$ and $C^{mat}_{q_{Cl_{id}}4}$, that satisfies the condition $\mathit{frac}_3 \times 1/\mathit{Rep} + \mathit{frac}_4 \times 1/\mathit{Rep} \geq \mathit{Rep}$. The verifier must choose a pair of consistency levels from the above matching set, such that resultant latency of q_3 and q_4 satisfies the respective deadlines, estimated with respect to the SLA, i.e., $L^j_{q_{Cl_{id}}3} \leq L^d_{q_{Cl_{id}}3}$ and $L^j_{q_{Cl_{id}}4} \leq L^d_{q_{Cl_{id}}4}$. Then it executes on the datastore with the consistency level predicted by the verifier. Upon completion of execution, it resets the lock column for *sum* to free and removes the corresponding entry from the registry. The failure handler (Section 3.2) listens for any error message returned by q_3. If q_3 fails with an exception like unavailability of replicas, failure handler rollbacks the effects of q_3 by restoring the value of *sum* to its old value It re-inserts q_3 to the operation queue, and updates the current deadline of q_3 with the delay due to the failed query. Once q_3 is removed from the registry, q_4 is released from the queue, and q_4 writes an entry the registry. According to the scheduling algorithm (Algorithm 1) requests for the lock to the key z at a particular instant of time. Once the lock for z is free, q_4 acquires the lock on z, and executes with the consistency level predicted by the verifier. The verifier uses the weakest precondition $P_{q_{Cl_{id}}4}$ to determine the weakest matching consistency levels $C^{mat}_{q_{Cl_{id}}4}$ and $C^{mat}_{q_{Cl_{id}}5}$, such that $\mathit{frac}_4 \times 1/\mathit{Rep} + \mathit{frac}_5 \times 1/\mathit{Rep} \geq \mathit{Rep}$, and $L^j_{q_{Cl_{id}}4} \leq L^d_{q_{Cl_{id}}4}$ and $L^j_{q_{Cl_{id}}5} \leq L^d_{q_{Cl_{id}}5}$. Once q_4 finishes execution, the scheduler sets the lock column to free and removes the entry from the registry. Finally, q_5 writes to the registry, and acquires the lock on z. It gets executed with the weakest possible consistency level from the matching set $C^{mat}_{q_{Cl_{id}}5}$, satisfying the SLA and producing the desired output, i.e., $z = 3$. If the application fails to satisfy the correctness condition, the updates due to the queries q_3, q_4, and q_5 are rolled back. The keys *sum* and z are restored to their old values, and a message is sent to the user informing that the application is being retried. The application retries q_3, q_4, and q_5 in the order of their invocation, inserting q_3 in the operation queue first. If the SLA is violated, the interface sends a failure message, along with the delay by which the SLA was overshoot. Consistify also asks the user if the user accepts the results despite the SLA violation, or if the application should be retried. In case of the latter

timestamp. $L^d_{q_{Cl_{id}}3}$ and $L^d_{q_{Cl_{id}}4}$ are the current deadlines for the query $q_{Cl_{id}}3$ and $q_{Cl_{id}}4$. For concurrent queries from multiple clients trying to access a given lock, the scheduler checks the registry entries to determine the waiting time $T^W_{Cl_{id}i}$ for each query. It also estimates the aggregate latency $L^{dep}_{Cl_{id}i}$ for the queries that are dependent on a given query. The scheduler then determines which client has the least amount of time left to satisfy its current deadline $L_{Cl_{id}}i^d$, i.e., it selects the query which produce the minimum value for the expression $L_{Cl_{id}}i^d - T^W_{Cl_{id}i} - L^{dep}_{Cl_{id}i}$, to request for the lock (Section 3.1).

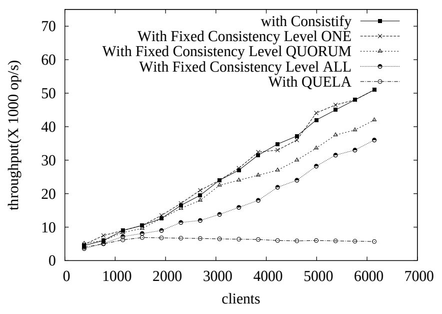
choice, the interface rollbacks the updates due to q_3, q_4, and q_5, and inserts the query q_3 in the operation queue.

6 Implementation

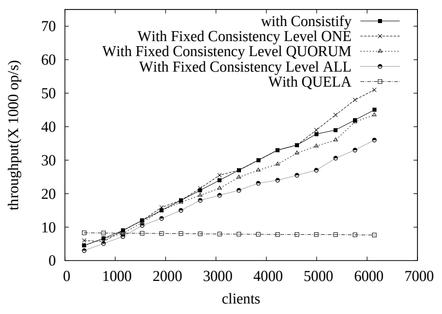
The Consistify interface is implemented as a Java middleware, using connection and accessor methods provided by the Cassandra driver core version 2.1.6 maintained by the Datastax community. Consistify uses 95 percentile latency estimates obtained by running YCSB [8] benchmark workloads with each possible consistency level on clusters comprising same number of nodes and replication factor. Consistify provides the developers with a high level API comprising a group of accessor methods that can be used to execute different storage operations on a quorum-based datastore while hiding the internal syntactic details of the underlying datastore. The Consistify API allows application developers the flexibility of developing the application without requiring them to possess any knowledge of the internal mechanisms of the underlying datastore. The developer passes the correctness condition and the SLA deadline to the API methods in the form of a first order logic statements. The Consistify interface parses the above statements, and determines the weakest possible consistency levels for the given sequence of operations. Applying the above consistency levels, the developer calls the accessor methods from the API to execute the given sequence of operations on the underlying datastore. The open source codebase for the Consistify interface can be found in the github repository: `https://github.com/ssidhanta/Consistify`. The methods in the Consitify API are of the following format. $< insert/read/update >$ $(< keyspace\ name >$ $, < columnfamily\ name >, < primary\ key\ name >, < primary\ key\ value >$ $, < column\ name >, < column\ value\ argument >, < client\ thread\ index >$ $, < current\ time >, < estimated\ latency\ for\ insert >, < consistency\ level >$ $, < client\ id\ , < invocation\ time >\ , < conectionobject >)$ For example, the correctness condition for the shopping cart application is given by the following first order logic relation. $AMT = \sum_{items} ITEM\_PRICE * QTY$. Also, we integrated the Consistify interface with PY-TPCC, a widely used [10] state-of-the-art open source python implementation of the TPC-C benchmark. The codebase for our TPC-C prototype can be found in the github repository: `https://github.com/ssidhanta/py-tpcc-master`.

6.1 Sample Client Applications and the Correctness Conditions Applied

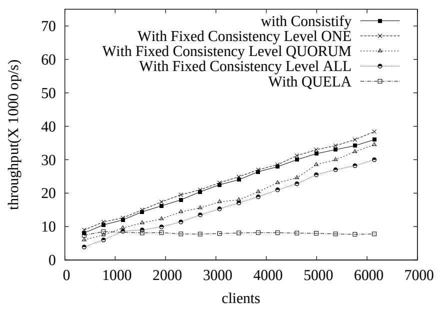
Here we give the implementation details, in Java, of the sample client applications used in our Evaluation section. We also give the correctness conditions given as input to the sample applications. The correctness conditions are given as first order logic relations that comprise some columns/keys accessed by the client application.



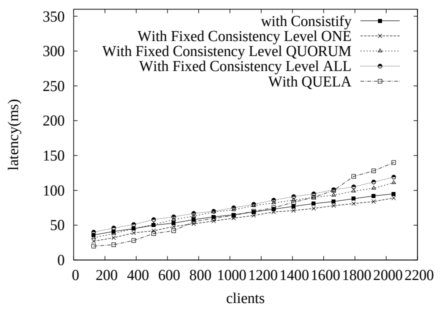
(a) Throughput vs No of Clients for Stocking Trading Application



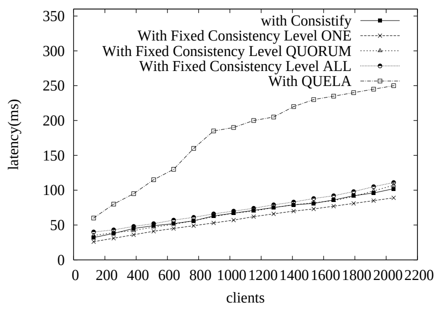
(b) Throughput vs No of Clients for Shopping Cart Application



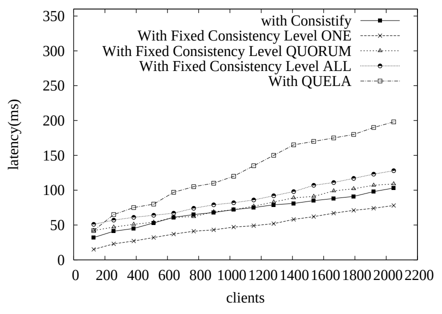
(c) Throughput vs No of Clients for Retail Store Application



(d) Latency vs No of Clients for Stock Trading Application

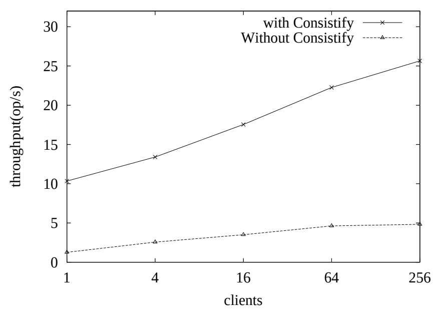


(e) Latency vs No of Clients for Shopping Cart Application

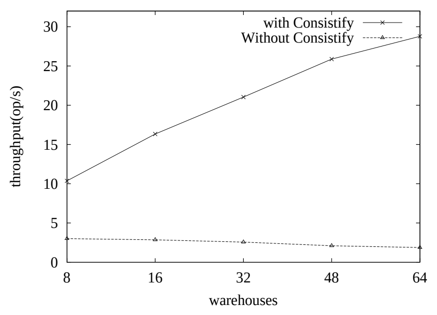


(f) Latency vs No of Clients for Retail Store Application

Fig. 3: Results With Consistify vs those With QUELA and those With Manually Chosen Fixed Consistency Level



(a) Throughput vs No of Clients for TPC-C



(b) Throughput vs No of Warehouses for TPC-C

Fig. 4: Results With TPC-C With and Without Consistify

Stock Trading The correctness condition for the Stock Trading application is given as the following first order logic relation.

$$\sum_{stocks} STOCK\_PRICE * STOCK\_NUM < \sum_{stocks} STOCK\_PRICE_{BUY} * STOCK\_NUM$$

The Java code for calling the Consistify API from the sample stock trading application is given below.

```
Interface intrface = new Interface();
Tuple tuple = intrface.callInterface(<correctness condition>,
    <cient application file name>,<client id argument>, <
    invocation time argument>,<current therad count>);
intrface.insert("stock","stock","STOCK_ID",<stock Id argument
    >, "STOCK_NUM",<number of stocks argument>,"STOCK_DESC",<
    stock description value>, "STOCK_PRICE", <stock price
    argument>, "", "", "", index, System.currentTimeMillis(),
    Interface.latency_insert,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
intrface.insert("stock","orders","ORDER_ID",<order id argument
    >, "STOCK_ID",<stock Id argument>,"STOCK_NUM",<number of
    stocks argument>, "STOCK_PRICE", <stock price argument>, "
    ", "", "", index, System.currentTimeMillis(),Interface.
    latency_insert,<consistency level argument>,<client id
    argument>, <invocation time argument>,tuple);
index++;
intrface.insert("stock","investors","INVESTOR_ID",<investor id
     argument>, "INVESTOR_NAME",<investor name argument>,"
    DEPOSIT",<deposit value argument>, "CREATE_DATE", <
    creation date argument>, "", "", "", index, System.
    currentTimeMillis(),Interface.latency_insert,<consistency
    level argument>,<client id argument>, <invocation time
    argument>,tuple);
index++;
String price=intrface.read("stock","orders","ORDER_ID",<order
    id argument>, index, System.currentTimeMillis(),Interface.
    latency_read,<consistency level argument>,<client id
    argument>, <invocation time argument>,tuple);
index++;
String price1=intrface.read("stock","investors","INVESTOR_ID"
    ,<investor id argument>, index, System.currentTimeMillis()
    ,Interface.latency_read,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
intrface.update("stock","orders","CONFIRMATION",<confirmation
    status argument>,"ORDER_ID", <order id argument>, "",
    index, System.currentTimeMillis(),Interface.latency_update
    , <consistency level argument>,<client id argument>, <
    invocation time argument>,tuple);
index++;
intrface.insert("stock","order_history","ORDER_ID",<order id
    argument>, "STOCK_ID",<stock id argument>,"INVESTOR_ID",<
    investor id argument>, "CREATE_DATE", <creation date
    argument>, "", "", "", index, System.currentTimeMillis(),
    Interface.latency_insert,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
```

Shopping Cart The correctness condition for the shopping cart application is given as the following first order logic relation.

$$AMT = \sum_{items} ITEM\_PRICE * QTY$$

The code for calling the Consistify API from the sample shopping cart application is given below.

```
Interface intrface = new Interface();
Tuple tuple = intrface.callInterface(<correctness condition>,
    <cient application file name>,<client id argument>, <
    invocation time argument>,<current therad count>);
intrface.insert("shopping_cart","catalog","ITEM_ID",<item id
    argument>, "ITEM_NAME",<item name argument>,"ITEM_DESC",<
    item description argument>, "STATUS", <cart status
    argument>, "", "", "", index, System.currentTimeMillis(),
    Interface.latency_insert,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
```

```
String price=intrface.read("shopping_cart","catalog","STATUS"
    ,<cart status argument>, index, System.currentTimeMillis()
    ,Interface.latency_read,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
intrface.insert("shopping_cart","cart","ITEM_ID",<item id
    argument>, "ITEM_DESC",<item description argument>,"QTY",<
    item quantity argument>, "CREATE_DATE", <creation date
    argument>, "", "", "", index, System.currentTimeMillis(),
    Interface.latency_insert,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
intrface.insert("shopping_cart","orders","ORDER_ID",<order id
    argument>, "ITEM_ID",<item id argument>,"QTY",<item
    quantity argument>, "CART_ID", <cart id argument>, "", "",
     "", index, System.currentTimeMillis(),Interface.
    latency_insert,<consistency level argument>,<client id
    argument>, <invocation time argument>,tuple);
index++;
intrface.insert("shopping_cart","invoice","INVOICE_ID",<
    invoice id argument>, "ORDER_ID",<order id argument>,"AMT"
    ,<invoice amount argument>, "PAID_STATUS", <paid status
    argument>, "", "", "", index, System.currentTimeMillis(),
    Interface.latency_insert,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
index++;
intrface.update("shopping_cart","invoice","PAID_STATUS",<paid
    status argument>,"INVOICE_ID", <invoice id argument>, "",
    index, System.currentTimeMillis(),Interface.latency_update
    , <consistency level argument>,<client id argument>, <
    invocation time argument>,tuple);
```

Retail Application The correctness condition for the shopping cart application is given as the following first order logic relation.

$PRICE \leq \sum itemsITEM\_PRICE * QTY$

The code for calling the Consistify API from the sample retail application is given below.

```
Interface intrface = new Interface();
Tuple tuple = intrface.callInterface(<correctness condition>,
    <cient application file name>,<client id argument>, <
    invocation time argument>,<current therad count>);
String price=intrface.read("consistify","orders","first", <
    first name argument>,index, System.currentTimeMillis(),
    Interface.latency_read,<consistency level argument>,<
    client id argument>, <invocation time argument>,tuple);
```

```
index++;
intrface.update("consistify","orders","last","price","first",
    <first name argument>,"QTY",<item quantity argument>, "
    ITEM_ID" ,<item id argument>,"ITEM_PRICE" ,<item price
    argument>,"",index, System.currentTimeMillis(),Interface.
    latency_update, <consistency level argument>,<client id
    argument>, <invocation time argument>,tuple);
index++;
String new_price=intrface.read("consistify","orders","first",<
    first name argument>,"ITEM_PRICE" ,<item price argument>,
    index, System.currentTimeMillis(),Interface.latency_read,<
    consistency level argument>,<client id argument>, <
    invocation time argument>,tuple);
index++;
```

7 Evaluation

We run our experiments on a geo-replicated testbed comprising 3 Amazon [5] c3.2xlarge ec2 instances, spread out across 2 ec2 regions, running Apache Cassandra 2.1.6 with a replication factor of 3, loaded on top of Ubuntu 13.10. Following the state-of-the-art [21], we discuss the results of experiments performed with three benchmark applications - stock trading, shopping cart, and online retail store. The client applications are invoked from remote c3.2xlarge ec2 instances running Ubuntu 13.10, which are not collocated with the servers. Figures 3(a), 3(b), 3(c), 3(d), 3(e), and 3(f) present the throughput (in terms of operations per second) and latency (in milliseconds) obtained against varying number of clients, with consistency levels chosen with Consistify, and with manually chosen fixed consistency levels, respectively. The results indicate that Consistify succeeds in producing increased observed throughput and lower observed latency, in contrast with fixed manually chosen consistency settings (like ONE, QUORUM, ALL in Cassandra). This can be attributed to the application of the weakest possible consistency setting in each case. With the exception of Quela [21], Consistify is the only system to allow client applications to enforce correctness conditions on top of quorum-based datastores. On top of that, Consistify can work under SLA deadlines unlike Quela. The results with the benchmark applications demonstrate that Consistify outperform the state-of-the-art, in terms of both obtained throughput, while also satisfying the latency SLAs. Specifically, the observed throughput with Consistify varies between 2 to 10 times the values reported by Quela [21], and the observed latency is also consistently less with Consistify, except for the stock trading application. The goal of Consistify is maximizing the throughput, rather than latency, while not violating the correctness conditions, under given SLA deadlines. Importantly, Consistify successfully satisfies imposed SLA deadlines (5 seconds per application), while Quela does not consider SLAs at all.

[5] Consistify is supported by an AWS in Education Research Grant award.

We also present the results with an open source implementation of the TPC-C benchmark over Apache Cassandra 2.1.6. Figure 4(a) presents the throughput measures (operations per second) against varying number of clients, while Figure 4(b) presents the throughput against varying number of warehouses, with and without consistency levels chosen with Consistify. Apart from Ashraf et. al. [1], Consistify is the first work that provides documented results of running TPC-C benchmark workloads over a quorum-based datastore, instead of a relational database. Figures 4(a) and 4(b) show that Consistify clearly outperforms the state-of-the-art [1]. The low throughput values with Cassandra, in contrast to typical relational databases, can be attributed to some reported unsolved issues with the TPC-C implementation, like intermittent job failures, especially with multiple clients and warehouses. However, fixing the above issues were out of the scope of this paper. Nevertheless, our results clearly beat (over 5 times) the only documented results of TPC-C over Cassandra [1].

7.1 Insights

We demonstrate that Consistify enables client applications to obtain throughput results as high as those obtained under eventual consistency guarantees, while avoiding the anomalies that may result with eventual consistency. The state-of-the-art correctness preserving tools, like Quela [21], use traditional approaches, like storing local snapshots, and merging the snapshots in regular intervals, or applying central database level locks. The snapshot approach introduces additional memory and latency overhead, while the locking technique is computationally expensive. Quela [21] specifically uses the lightweight transactions primitive of Cassandra 2.x, which is unreliable and expensive because of additional Paxos roundtrips [14]. Hence, none of the state-of-the-art approaches can enable client applications to work under stringent deadline, or work under computationally resource constrained environments (like laptops and commodity machines). On the other hand, Consistify uses lightweight data structures to determine the dependency relations among the storage operations comprising a client application. Such datastructures are alive only during the lifespan of the client application, hence do not add to the resource usage of the system. Instead of central database-level locks, Consistify uses a registry of control flags to temporarily lock the columns being accessed by a given operation for providing serializability guarantees under concurrent client execution. Control flags are similar in nature to mutex variables, which are used in the operating system domain to control concurrent process execution. The registry is maintained in a separate control plane in the distributed datastore itself, hence do not require any additional storage or sharing mechanism. Each control flag in the registry is accessed only during the actual invocation of the operation, hence the latency overhead for concurrency control is checked to a bare minimum. Since the registry control function is handled in parallel with the execution of the other operations, the throughput is maximized.

8 Related Work

In practice eventual consistency is preferred over strong consistency in scenarios where the system must maintain availability during network partitions [4, 23], or when the application is latency-sensitive and able to tolerate occasional inconsistencies [6, 11]).A large body of research deals with the problem of supporting various forms of stronger-than-eventual consistency in scalable storage systems and databases. The state machine replication paradigm achieves the strongest possible form of consistency by physically serializing operations [15]. Lamport's Paxos protocol is a quorum-based fault-tolerant protocol for state machine replication [16]. Mencius improves upon Paxos by ensuring better scalability and higher throughput under high client load using a rotating coordinator scheme [18]. A number of scalable fault-tolerant storage systems have been constructed using variations on Paxos [20, 5, 13, 9].

Relatively few systems [24, 22, 3, 21] provide mechanisms for fine-grained control over consistency. Pileus and Tuba [22, 3] are the only systems that come close to providing fine grained consistency tuning using SLAs. But, instead of predicting, these systems perform actual trials on the system, and select the consistency level corresponding to the SLA row that produces minimum resultant utility, based on the trial outcomes. The trial-based technique can produce unreliable results due to the unpredictable parameters like network conditions and workload that affect the observed latency and staleness. Thus, predictions based on the outcomes of the trial phase may be unsuitable in the actual running time of the operation. Sivaramakrishnan et. al. [21] use a static analysis approach to determine the weakest consistency level that satisfies the correctness conditions declared in the contract. It requires the users to have knowledge of a declarative language for specifying the contract, and to accurately specify the correctness rules in the contract. Also, it does not explicitly consider the latency as an SLA parameter, and cannot dynamically adapt to varying workload and network state. Li et. al. [17] applies static analysis for automated consistency tuning for relational databases.

9 Conclusions

Consistify tunes the underlying datastore with the provably valid weakest possible consistency settings under the given SLA deadline and the correctness condition. Consistify provides an abstraction layer that allows application developers to develop client applications on top of a quorum-based datastore, without requiring the developers to have any expertise of the mechanisms of the underlying datastore. Consistify allows the client applications to work under different consistency guarantees, determined by the user-specified correctness condition, on top of the same underlying datastore, under given SLA deadline. The Consistify interface upgrades the datastore such that it is possible to enforce the above correctness conditions on the given datastore, without additional memory, storage, or latency overhead for maintaining snapshots, applying central locks, or CAS. Experimental results demonstrate that Consistify exceeds the observed performance of the state-of-the-art systems by factor of 2 to 10.

References

1. A. Aboulnaga. High Availability for Database Systems in Cloud Computing Environments. `http://research.microsoft.com/apps/video/default.aspx?id=156491`, 2016. [Online; accessed 20-January-2016].
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 367–381, Broomfield, CO, Oct. 2014. USENIX Association.
4. K. Birman and R. Friedman. *Trading Consistency for Availability in Distributed Systems*. Cornell University. Department of Computer Science, 1996.
5. W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.
6. E. A. Brewer. Towards robust distributed systems (Invited Talk). In *Proc. of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2000.
7. B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.
8. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.
9. J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.
10. F. Cruz, F. Maia, M. Matos, R. Oliveira, J. a. Paulo, J. Pereira, and R. Vilaça. Met: Workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.
11. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
12. M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, Mar. 1994.
13. T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Proc. of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

14. A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
15. L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558, 1978.
16. L. Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.
17. C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.
18. Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.
19. C. Meiklejohn. Riak PG: Distributed process groups on dynamo-style distributed storage. In *Proc. of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 27–32, New York, NY, USA, 2013. ACM.
20. J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243, 2011.
21. K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. *SIGPLAN Not.*, 50(6):413–424, June 2015.
22. D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.
23. D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 172–182, 1995.
24. H. Yu and A. Vahdat. Building replicated internet services using TACT: A toolkit for tunable availability and consistency tradeoffs. In *WECWIS*, pages 75–84, 2000.