Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants

Valter Balegas, Diogo Serra, Sérgio Duarte
Carla Ferreira, Marc Shapiro*, Rodrigo Rodrigues†, Nuno Preguiça
NOVA LINCS, FCT, Universidade NOVA de Lisboa †INESC-ID Instituto Superior Técnico, Universidade de Lisboa
*Inria Paris-Rocquencourt & Sorbonne Universités, UPMC, LIP6

***Abstract*—Geo-replicated databases often offer high availability and low latency by relying on weak consistency models. The inability to enforce invariants across all replicas remains a key shortcoming that prevents the adoption of such databases in several applications. In this paper we show how to extend an eventually consistent cloud database for enforcing numeric invariants. Our approach builds on ideas from escrow transactions, but our novel design overcomes the limitations of previous works. First, by relying on a new replicated data type, our design has no central authority and uses pairwise asynchronous communication only. Second, by layering our design on top of a fault-tolerant database, our approach exhibits better availability during network partitions and data center faults. The evaluation of our prototype, built on top of Riak, shows much lower latency and better scalability than the traditional approach of using strong consistency to enforce numeric invariants.**

I. Introduction

Scalable cloud databases with a key-value store interface have emerged as the platform of choice for providing online services that operate on a global scale [11], [9], [7]. In this context, a common technique for improving the user experience is geo-replication [9], [7], i.e., maintaining copies of application data and logic in multiple data centers scattered across the globe. This decreases the latency for handling user requests by routing them to a nearby data center, but at the expense of resorting to weaker data consistency guarantees to avoid costly coordination among replicas.

When executing under such weaker consistency models, applications have to deal with concurrent operations. A common approach is to rely on a *last writer wins* strategy [15], [11], but this can lead to lost updates. To address this problem, some databases include specific reconciliation support for some data types, such as counters in Cassandra and DynamoDB, and CRDTs [18] in Riak.

Still, these approaches are unable to enforce invariants across all replicas. For example, it is impossible to enforce numeric invariants (e.g., $x \geq K$), which previous works have shown to be central for maintaining application correctness [14]. This prevents the adoption of such databases in many applications, such as virtual wallets in games, or management of stocks in e-commerce applications and ticket reservations. In this paper we show how to extend eventually consistent cloud databases for enforcing numeric invariants.

Maintaining this type of invariants would be trivial in systems that offer strong consistency guarantees, namely those that serialize all updates [14], [8]. The problem with these systems is that they require coordination among replicas, leading to an increased latency and reduced fault tolerance. In contrast, our approach builds on the key idea of escrow transactions [16], which is to partition the difference between the current value of a numeric variable and the bound to be enforced among existing replicas. These parts are distributed among replicas, who can locally execute operations that do not exceed their allocated part without contacting other replicas.

In this paper, we present the design of a middleware that overcomes a number of important limitations that exist in previous works that build on the same ideas. First, in contrast to previous escrow based approaches, ours includes no central authority and is totally asynchronous. To this end, we propose a novel replicated data type [18], the *Bounded Counter*, to maintain the information about the escrow each replica holds. Second, we layer the management of *Bounded Counter*s on top of an eventually consistent cloud database. Thus, our design inherits the fault tolerance properties of the underlying database and exhibits better availability than systems that use strong consistency, during network partitions and data center faults. Finally, our middleware combines caching with operation batching, thus improving write throughput without reducing the fault tolerance properties of the system.

The evaluation of our prototype, running on top of Riak, shows that: 1) when compared to using strong consistency, our approach can enforce invariants without paying the latency price for replica coordination, which is considerable for all but the local clients; 2) when compared to using weak consistency, our optimizations lead to higher throughput with a very small increase in latency, while guaranteeing that invariants are not broken.

The remainder of the paper is organized as follows. Section II overviews our approach; Section III introduces the *Bounded Counter* CRDT; Section IV presents our middleware that extends Riak with numeric invariant preservation; Section V evaluates our prototypes; Section VI discusses related work; and Section VII concludes the paper.

II. System Model

We target a typical geo-replicated scenario, with copies of application data and logic replicated in multiple data centers (DCs) scattered across the globe. End clients contact the closest DC for executing application operations. We consider that system processes are connected by an asynchronous network and assume that processes may fail by crashing. A crashed process may either remain crashed forever, or recover with its persistent memory intact.

System API: In addition to *get(key)* and *put(key, value)* operations to access common objects, our middleware provides the following operations to manipulate *Bounded Counter* objects:
(i) *create(key, type, bound)*, creates a new *Bounded Counter* with the given *key*, constraint *type* ($\geq, \leq$) and *bound* – e.g., *create('A', '$\geq$', 10)* creates a counter with initial value 10 that enforces constraint $A \geq 10$;
(ii) *value(key)*, returns the current value of counter *key*;
(iii) *inc(key, value, remote)* and *dec(key, value, remote)*, update the counter if it is known that the change will not break the invariant, with the *remote* flag allowing to request contacting remote nodes if necessary. Update operations return *success* if they succeed or *error* otherwise.

Consistency Guarantees: We build our middleware on top of an eventually consistent database, extending the underlying guarantees with invariant preservation for counters. In particular, the eventual consistency model means that the outcome of each operation reflects the effects of only the subset of operations that have already been executed by the replica that the client has contacted. However, for each operation that successfully returns at a client, there is a point in time after which its effect becomes visible to every operation that is invoked after that time, i.e., operations are eventually executed by all replicas.

In terms of the invariant preservation guarantee, our system guarantees that the value of the counter never violates the bounds specified by the invariant, neither *locally* nor *globally*. By locally, this means that the subset of operations seen by the replica must obey:
lower bound $\leq$ initial value $+\sum inc - \sum dec \leq$ upper bound.
By globally, this means that, at any instant in the execution of the system, when considering the union of all the operations executed by every replica, the same bounds must hold.

Note that the notion of causality is orthogonal to our design, in the sense that if the underlying storage system offers causal consistency, then we also provide numeric invariant-preserving causal consistency.

Enforcing Numeric Invariants: To enforce numeric invariants, our design borrows ideas from the escrow transactional model [16]. The key idea is to consider the difference between the value of a counter and its bound as a set of rights to execute operations. For example, in a counter, n, with initial value $n = 40$ and invariant $n \geq 10$, there are 30 rights to execute decrement operations. Executing *dec(5)* consumes 5 of these rights. Executing *inc(5)* creates 5 new rights. In this model, these rights can be split among the replicas of the counter – e.g., if there are 3 replicas, each replica can be assigned 10 rights. If the rights needed to execute some operation exist in the local replica, the operation can safely execute locally, knowing that the global invariant will not be broken – in the previous example, if the decrements of each replica are less or equal to 10, it follows that the total number of decrements does not exceed 30, and therefore the invariant is preserved. If not enough rights exist, then either the operation fails or additional rights must be obtained from other replicas.

Our approach encompasses two components that work together to achieve the goal of our system: a novel data structure, the *Bounded Counter* CRDT, to maintain the necessary information for locally verifying whether it is safe to execute an operation or not (Section III); and a middleware to manipulate instances of this data structure, which are persistently stored in the underlying cloud database (Section IV).

```
1:  payload integer[n][n] R, integer[n] U, integer min
2:      initial [[0,0,...,0], ..., [0,0,...,0]], [0,0,...,0], K
3:  query value () : integer v
4:      v = min + Σ_{i∈Ids} R[i][i] − Σ_{i∈Ids} U[i]
5:  query localRights () : integer v
6:      id = repId()                          %Id of the local replica
7:      v = R[id][id] + Σ_{i≠id} R[i][id] − Σ_{i≠id} R[id][i] − U[id]
8:  update increment (integer n)
9:      id = repId()
10:     R[id][id] = R[id][id] + n
11: update decrement (integer n)
12:     pre-condition localRights() ≥ n
13:     id = repId()
14:     U[id] = U[id] + n
15: update transfer (integer n, replicaId to): boolean b
16:     pre-condition b = (localRights() ≥ n)
17:     from = repId()
18:     R[from][to] := R[from][to] + n
19: update merge (S)
20:     R[i][j] = max(R[i][j], S.R[i][j]), ∀i, j ∈ Ids
21:     U[i] = max(U[i], S.U[i]), ∀i ∈ Ids
```

Fig. 1: *Bounded Counter* for invariant *greater or equal to K*.

III. Design of Bounded Counter CRDT

This section presents the *Bounded Counter*, a CRDT that maintains information for enforcing numeric invariants without requiring coordination for most executions of operations.

A. CRDT Basics

Conflict-free replicated data types (CRDTs) [18] are a class of distributed data types that allow replicas to be modified without coordination, while guaranteeing that replicas converge to the same correct value after all updates are propagated and executed in all replicas.

In this work, we adopted the state-based model of CRDTs, as we built our work on top of a key/value store (KV-Store) that synchronizes replicas by propagating the state of the database objects. In this model, an operation submitted in a given site executes in the local replica. Updates are then propagated among replicas in peer-to-peer interactions, where a replica r_1 propagates its state to another replica r_2, which merges its local state with the received state, by executing the *merge()* operation.

It has been proven that a sufficient condition for guaranteeing the convergence of the replicas of state-based CRDTs is that the object conforms the properties of a monotonic semi-lattice object [18], in which: *(i)* The set S of possible states forms a semi-lattice ordered by $\leq$; *(ii)* The result of merging state s with remote state s' is the result of computing the LUB of the two states in the semi-lattice of states, i.e., $merge(s, s') = s \sqcup s'$; *(iii)* The state is monotonically non-decreasing across updates, i.e., for any update u, $s \leq u(s)$.

R	r_1	r_2	r_3	U
r_1	30	10	10	5
r_2	0	1	0	4
r_3	0	0	0	2

Limit value (min): 10
Current Value: 20
Local rights:
$r_1 = 5, r_2 = 7, r_3 = 8$

Fig. 2: Example of the state of *Bounded Counter* for maintaining the invariant *greater or equal to 10*.

B. Bounded Counter CRDT

We now detail the *Bounded Counter*, a CRDT for maintaining the invariant *greater or equal to K*. The pseudocode is presented in Figure 1.

***Bounded Counter* state:** The *Bounded Counter* maintains the limit value K and information about the rights each replica holds. For a system with n replicas, this information is stored in: a matrix R, where entry $R[i][j]$ keeps the rights transferred from replica i to replica j; and in a vector U, where $U[i]$ keeps the rights consumed by replica i.

Operations: An *increment* executed at r_i updates the number of increments for r_i by updating the value of $R[i][i]$. This operation is safe and can always execute locally.

A *decrement* executed at r_i updates the number of decrements for r_i by updating the value of $U[i]$. This operation can only execute if r_i holds enough rights locally before executing the operation, otherwise the operation fails.

The rights of replica r_i, returned by function *localRights*, are given by adding the local increments $R[i][i]$ to the transfers from other replicas to r_i, given by $\sum_{j:j\neq i} R[j][i]$, subtracting the transfers from r_i to other replicas, $\sum_{j:j\neq i} R[i][j]$, and subtracting the local decrements $U[i]$.

Figure 2 shows an example of a *Bounded Counter* for the invariant *greater or equal to 10*. The initial value of the counter is the bound of the constraint, 10. Replicas r_1, r_2 and r_3 have incremented the counter by 30, 1 and 0 units, respectively, as shown in the diagonal of R. The current value of the counter is given by adding to the limit, the increments performed in every replica, $\sum_i R[i][i]$, and subtracting the decrements, $\sum_i U[i]$, as represented in the grey cells. The operation *transfer* transfers rights from r_i to some other replica r_j, by increasing the value recorded in $R[i][j]$. This operation can only execute if enough local right exist. In the example of Figure 2, transfers of 10 rights from r_1 to each of r_2 and r_3 are recorded in the values of $R[1][2]$ and $R[1][3]$.

The *merge()* operation is executed during peer-to-peer synchronization, when a replica receives the state of a remote replica. The local state is updated by just taking, for each entry, the maximum of the local and the received value.

In a companion technical report [4], we prove that the *Bounded Counter* is a CRDT and that the data structure ensures invariant maintenance in the presence of concurrent updates in different replicas. A TLA proof of correctness is also available.

Extensions: The exact same logic can be applied to preserve invariants of the form *less or equal to K*: Rights represent the possibility of executing *increment* operations instead of *decrement* operations. The specification of the data type is changed accordingly.

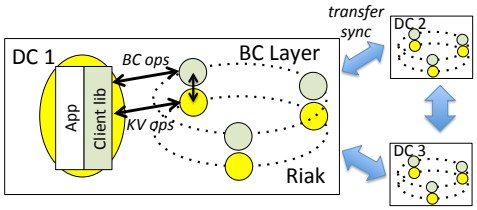

Fig. 3: Middleware for deploying *Bounded Counters*.

Some applications may require two bounds for a counter, e.g., *greater or equal to* K_0 and *less or equal to* K_1. A *Bounded Counter* can maintain an invariant of that form by combining the information of two *Bounded Counter*s in one object, similarly to what is done to specify a PN-Counter from two P-Counters [18]. Some expressions might involve constraints over multiple counters. With the current prototype, the only way to implement these is to store them in a single object, but we do not support it in the current interface.

In general, the approach used for *Bounded Counter*s can be applied to other data types that support escrow [22].

Optimizations: The state of *Bounded Counter* has complexity $O(n^2)$, for n logical replicas. In practice, the impact of this is expected to be small as the number of data centers in common deployments is typically small and each DC will typically hold a single logical replica. In a technical report [4] we show how to lower the space complexity of *Bounded Counters* to $O(n)$.

IV. Middleware for Enfocing Numeric Invariants

We now present a middleware, depicted in Figure 3, that uses *Bounded Counter*s to extend cloud databases with numeric invariants. In each DC, our system is composed by a set of middleware nodes and an underlying key-value store to persistently store data. Operations on regular objects execute directly in the key-value store. Operations on counters are handled by middleware nodes, with client requests routed to a specific node using a DHT communication substrate.

In our prototype, we use *riak_core* [10] as the DHT communication substrate and Riak 2.0, a key-value store inspired in Dynamo [9], as the underlying storage system. Riak 2.0 also includes a conditional write mode, where a write from a client fails if there has been a concurrent write since the client's previous operation. Our middleware uses this mechanism to serialize the execution of operations for each counter in each replica. We deploy a logical replica of the *Bounded Counter* per DC, which is replicated in a quorum of nodes by Riak. An operation in a counter is sent to the DHT node responsible for the counter. The DHT node executes the operation by reading the counter from Riak, executing the operation and writing back the new value, using the conditional write mechanism. The operation only succeeds if it is safe, i.e., if the local replica holds enough rights to guarantee the invariant is preserved. By using the conditional write mechanism, we guarantee that operations in each *Bounded Counter* execute sequentially without requiring any guarantees from the DHT. For example, if during a reconfiguration, concurrent requests to the same counter are sent to two different nodes, our approach is still safe as one of the operations will fail when writing to Riak.

Since Riak does not geo-replicate keys marked as strongly consistent, our middleware is responsible for replicating

Bounded Counters across DCs. To this end, each DHT node periodically propagates modified *Bounded Counters* to the remote DCs. When the payload is delivered on the remote DC, it is merged with the local state. This strategy batches a sequence of local operations on a single key and propagates them in a single update, saving bandwidth and processing.

Transferring Rights: Our middleware exchanges rights between replicas in two situations. First, when an operation cannot execute in a replica and the application has specified that remote replicas should be used. In this case, the DHT node executing the operation requests a transfer from a remote DC. To this end, it sends a message to a node in the remote data center, so that it executes a *transfer* operation in the *Bounded Counter*. Second, replicas proactively exchange rights in the background periodically to balance the rights assigned to each replica. These mechanisms are detailed in a separate document [4].

Fault tolerance: We now analyze how our middleware designs provide fault tolerance building on the fault tolerance properties of the underlying cloud database.

The cloud database is assumed to have sufficient internal redundancy to never lose its state in a DC. However, a failure in a node of the middleware layer may cause the DHT to reconfigure, with the possibly that two nodes temporarily accept requests for the same key. This does not affect correctness as we rely on conditional writes to guarantee that operations of each counter are serialized.

During a network partition, rights can be used in both sides of the partition – the only restriction is that it is impossible to transfer rights between any two nodes in different partitions. If an entire DC becomes unavailable, only the rights owned by the unreachable DC become temporarily unavailable. This contrasts with state-of-the-art strong consistency protocols [12], which can only serve requests if at least a majority of replicas (or a primary) is reachable. In our approach, any replica can serve requests if it owns enough rights or if it can gather the needed rights from reachable replicas.

Improving the performance of the middleware: Our prototype includes a number of optimizations to improve its efficiency. The first optimization is to cache *Bounded Counters* on the DHT nodes. This allows us to avoid reading the counter, when it is already in cache. Second, under high contention in a *Bounded Counter*, the design described so far is not very efficient, since an operation must complete before the next operation starts being processed. In particular, since processing an update requires writing the modified *Bounded Counter* back to the Riak database to ensure durability, each operation can take a few milliseconds to complete. To improve throughput, while the write to Riak is taking place, the requests received by the DHT node are processed using the cached counter. The system still writes the batched updates to storage before replying to the waiting clients, but this strategy allows to execute a single write for multiple requests. Our evaluation shows that this strategy improves the throughput of the system by orders of magnitude.

V. Evaluation

We evaluated experimentally our prototype to address the following main questions. (i) How much overhead is introduced by our middleware? (ii) What is the throughput and latency for different levels of contention? (iii) What is the latency when the value is close to the invariant bounds?

A. Configurations and Setup

In the experiments, we compare our middleware, *BC*, with the following configurations:

Weakly Consistent Counters (Weak). This configuration uses Riak 2.0 Enterprise Edition (EE), with native counters running under weak consistency. Native counters handle conflicts automatically inside the database layer. The native geo-replication mechanism of Riak EE is used.

Strongly Consistent Counters (Strong). This configuration runs a Riak 2.0 Community Edition database (for using conditional writes) in a single DC, serving local and remote requests. Updating a counter uses the conditional write mechanism of Riak for enforcing serializability, only succeeding if no concurrent write has completed.

Our experiments comprised 3 Amazon EC2 DCs distributed across the globe. The average latency between DCs is: US-East–Us-West, 80 ms; Europe (EU-West)–US-East, 96 ms; Europe–US-Wast, 160 ms. In each DC, we use three m1.large machines with 7.5GB of memory for running the database servers and server-based middleware and three m1.large machines for running the clients.

Data is fully geo-replicated in all DCs, with clients accessing the replicas in the local DC. Riak operations use a quorum of 3 replicas for writes and 1 replica for reads. In *Strong*, geo-replication is not used, data is stored in the US-East DC, which minimizes the latency for remote clients.

B. Single Counter

We first evaluate performance under high contention. To this end, we use a single counter initialized to a value that is large enough to never break the invariant. Clients execute 20% of increments and 80% of decrements in a closed loop with a think time of 100 ms. Each experiment runs for two minutes after the initialization of the database. The load is controlled by tuning the number of clients running in each experiment, with clients evenly distributed among the client machines.

Throughput vs. latency: Figure 4 presents the variation of the throughput vs. latency values as more operations are injected in the system.

The results of *Strong* show that throughput quickly starts degrading when load increases. This occurs because when more clients try to submit operations to a single DC they increase the interference, which prevents the conditional write from succeeding. We also observe that *Strong* exhibits the higher latency values which occurs because requests are all redirected to a single DC which is remote for $^2/_3$ of the clients.

In comparison to *Strong*, the throughput of *Weak* is much larger and it does not degrade when increasing the load – after reaching the maximum throughput, increasing the load just leads to an increase in latency. The much higher throughput of the middleware solution is due to the batching mechanism of *BC*, which batches a sequence of updates into a single write to storage. To prove this hypothesis, we ran the same experiment, turning off the batching and writing every update in Riak,

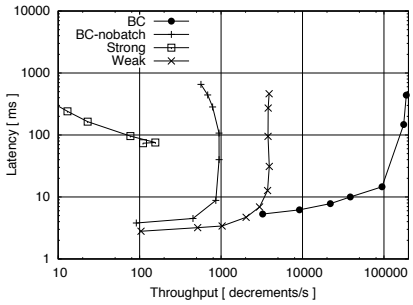

Fig. 4: Throughput vs. latency with a single counter.

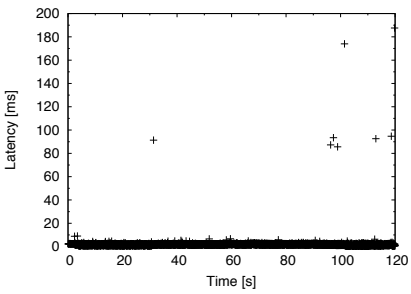

Fig. 5: Latency of each operation over time for BC.

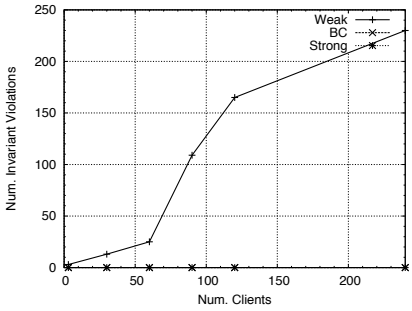

Fig. 6: Decrements executed in excess, violating invariant.

TABLE I: Latency of operations with a single counter.

Median (Max) latency (ms)	Weak	Strong	BC
US-East	2 (7)	172 (180)	4 (9)
US-West	2 (7)	169 (187)	8 (13)
Europe	2 (8)	5 (9)	5 (11)

BC-nobatch. In this case, we can observe that the throughput is much lower than *Weak*, as the middleware introduces an additional communication step and executes operations in sequence. The same approach for batching multiple operations into a single Riak write could be used with other configurations, such as *Weak*, to improve their scalability.

Latency under low load: Table I presents the median and maximum latency experienced by clients in different regions under low load. As expected, the results show that for *Strong*, remote clients experience high latency, while local clients are fast. It also shows that our middleware introduces an overhead of about about 2 ms when compared with *Weak*, which is justified by the additional communication steps.

Effects of exhausting rights: In this experiment we evaluate the behavior of our middleware when the value of the counter approaches the limit and contention for the last available rights rises. We initialize the counter with the value 6000 and 5 clients execute decrement operations until all rights are consumed. Figure 5 shows that most operations have low latency, with a few peaks of high latency whenever a replica needs to obtain additional rights. The number of peaks is small because most of the time the proactive mechanism for exchanging rights is able to provision a replica with enough rights before all local rights are consumed. We see these peaks more frequently near the end of the experiment, because there are less resources available and they might be temporarily exhausted. When all resources are consumed, replicas stop requesting rights and operations fail locally.

Invariant Preservation: To evaluate the severity of the risk of invariant violation, we computed how many decrements in excess were executed with success in the different solutions. We run the same experiment as before, but vary the number of clients. Figure 6 shows that *Weak* is the only configuration that experiences invariant violation. The operation for decrementing consists in reading the counter, checking if the value is greater than the limit and executing a decrement. The decrement operation is not atomic and because of this, multiple decrements can execute concurrently considering the same read value. This effect increases with the number of clients and concurrent updates.

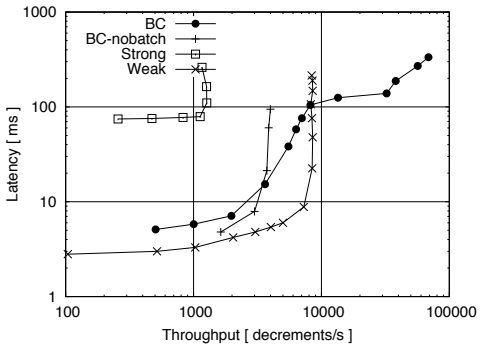

Fig. 7: Throughput vs. latency with multiple counters.

C. Multiple Counters

To evaluate how the system behaves in the common case where clients access to multiple counters, we ran the experiment of Section V-B with 100 counters. For each operation, a client selects the counter to update randomly with uniform distribution. The results presented in Figure 7 show that *Strong* now scales to a larger throughput. The reason for this is that by increasing the number of counters, the number of concurrent writes to the same key is lower, leading to a smaller number of failed operations. Additionally, when the maximum throughput is reached, the latency degrades but the throughput remains almost constant.

The *Weak* configuration scales up to a much larger value (9K decrements/s compared with 3K decrements/s for a single counter). As each Riak node includes multiple virtual nodes, when using multiple counters the load is balanced among them – enabling multi-core capabilities to process multiple requests in parallel (whereas with a single node, a single virtual node is used, resulting in requests being processed sequentially).

The results show that *BC* has a low latency (close to that of *Weak*) as long as the number of writes can be handled by Riak's conditional write mode in a timely manner. In contrast with the experiment with a single counter, Riak's capacity is shared among all the keys, each contributing with writes to Riak. Therefore, as the load increases, writing batches to Riak will take longer to complete and contribute to accumulate latency sooner than in the single key case. Nevertheless, batching still allows multiple client requests to be processed per each Riak operation, leading to a better throughput. The maximum throughput even surpasses the results for the *Weak* configuration.

The results for *BC-nobatch*, where each individual update is written using one Riak operation, can be seen as the worst case of our middleware, in which the batching had no effect. Still, since all *BC* operations are local to a given DC and access only a quorum of Riak nodes, one can expect that increasing the local cluster's capacity should have a positive effect both on latency and throughput.

VI. Related work

Many cloud databases supporting geo-replication have been developed in recent years. Several of them [9], [15], [1], [11], [6], [20] offer variants of eventual/weak consistency where operations return immediately once executed in a single DC. For some applications, strong consistency is necessary to ensure correctness [8]. To avoid the cost of strong consistency for all operations, some systems support both weak and strong consistency for different operation types or objects [14], [7], [20], [6]. In contrast, our work extends eventual consistency with numeric invariants, aiming to keep latency low for all operations.

Bailis et al. [2] examine which invariant of database systems can be enforced without coordination. Indigo [3] extends this approach by providing mechanisms to enforce generic invariants without coordination in most cases. In this work, the focus is on the implementation of a middleware that can be used on top of production databases to provide numeric invariants. We use Riak as a proof of its applicability and show experimentally how to enhance the system's performance by making good use of CRDTs.

Escrow transactions [16], initially proposed for increasing concurrency of transactions in single databases, have also been used for supporting disconnected operation in mobile computing environments either relying on centralized [17], [22] or peer-to-peer [19] protocols for escrow distribution. The demarcation protocol [5] enforces numeric invariants across multiple objects, located in different nodes. Additionally, it shows how to encode other invariants, such as referential integrity, using numeric invariants, which could also be explored in our work. Our work combines convergent data types [18] with ideas from these systems to provide a decentralized approach with replicated data that offers both automatic convergence and invariant preservation with no central authority. Additionally, we describe, implement and evaluate how such solution can be integrated into existing cloud databases.

VII. Conclusion

This paper presents a middleware to extend eventually consistent cloud databases for enforcing numeric invariants. Our design allows most operations to complete within a single DC by moving the necessary coordination outside of the critical path of operation execution. Additionally, our design exhibit a high degree of fault tolerance, by building on the high availability of the underlying database. Thus, we have shown how to combine the benefits of eventual consistency, low latency and high availability, with those of strong consistency, enforcing global numeric invariants. The evaluation of our prototype shows that our middleware has competitive performance when compared with Riak's native weak consistency mechanism where invariants can be compromised.

Acknowledgments

This research is supported in part by EU FP7 SyncFree project (609551), FCT/MCT Phd grant SFRH/BD/87540/2012, SwiftComp project (PTDC/ EEI-SCR/ 1837/ 2012) and NOVA LINCS (UID/CEC/04516/2013). The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant.

References

[1] Almeida, S., Leitão, J. a., and Rodrigues, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. EuroSys '13* (2013).

[2] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination Avoidance in Database Systems. In *Proc. VLDB'15* (2015).

[3] Balegas V., Duarte S., Ferreira C., Preguiça N., Rodrigues R., Najafzadeh M., Shapiro M. Putting Consistency Back into Eventual Consistency. In *Proc. EuroSys '15* (2015).

[4] Balegas, V., Serra, D., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N. M., Shapiro, M., and Najafzadeh, M. Extending eventually consistent cloud databases for enforcing numeric invariants. *CoRR abs/1503.09052* (2015).

[5] Barbará-Millá, D., and Garcia-Molina, H. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal 3* (July 1994).

[6] Basho. Riak. *http://basho.com/riak/*. Accessed Apr/2015.

[7] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1* (Aug. 2008).

[8] Corbett, J. C., Dean et al. Spanner: Google's globally-distributed database. In *Proc. OSDI'12* (2012).

[9] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available Key-Value Store. In *Proc. SOSP '07* (2007).

[10] Klophaus, R. Riak core: Building distributed applications without shared state. In *CUFP '10* (2010).

[11] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[12] Lamport, L. The Part-time Parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[13] Lamport, L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (May 1994), 872–923.

[14] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. OSDI'12* (2012).

[15] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. SOSP '11* (2011).

[16] O'Neil, P. E. The escrow transactional method. *ACM Trans. Database Syst. 11*, 4 (Dec. 1986), 405–430.

[17] Preguiça, N., Martins, J. L., Cunha, M., and Domingos, H. Reservations for conflict avoidance in a mobile database system. In *Proc. MobiSys '03* (2003).

[18] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proc. SSS '11* (2011).

[19] Shrira, L., Tian, H., and Terry, D. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *Proc. Middleware '08* (2008).

[20] Sivasubramanian, S. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *Proc. SIGMOD '12* (2012).

[21] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proc. SOSP '11* (2011).

[22] Walborn, G. D., and Chrysanthis, P. K. Supporting semantics-based transaction processing in mobile database applications. In *Proc. SRDS '95* (1995).