Bolt-On Convergence in Mergeable Replicated Data Types

Gowtham Kaki
University of Colorado Boulder
USA

Prasanth Prahladan
University of Colorado Boulder
USA

Nicholas Lewchenko
University of Colorado Boulder
USA

Abstract

Conflict-free Replicated Data Types (CRDTs) are popular building blocks of distributed applications. CRDTs guarantee eventual convergence of the replicated state provided that the updates to the state are defined in terms of commutative effects. However, non-commutative operations are common among ordinary data types, and lifting such an ordinary type to a CRDT involves non-trivial re-engineering of its internal state and operations to support commutative effects. Furthermore, substantial proof effort need to be expended to show that the resultant data type is indeed a CRDT.

In this paper, we describe an alternative approach to promote ordinary data types to convergent replicated types without the need for complex re-engineering or type-specific algebraic reasoning. Our approach extends Mergeable Replicated Data Types (MRDTs) with a runtime that enforces convergence by orchestrating only the *well-formed* distributed executions. Notably, such run-time enforcement impacts neither the latency of data type operations nor the availability of the overall system. Our approach thus confers the benefits of CRDTs without the need to constrain implementations to satisfy algebraic laws such as commutativity. We describe Quark– an implementation of the aforementioned runtime, and present an empirical evaluation involving a collaborative editing case study.

CCS Concepts: **• Computing methodologies → Distributed programming languages**; **• Computer systems organization → Availability**; **• Software and its engineering** → *Formal software verification.*

Keywords: Replication, MRDTs, CRDTs, Convergence, Runtime, Git, Version Control

1 Introduction

Distributed applications often define their state in terms of *Conflict-free Replicated Data Types* (CRDTs) that are specially engineered to reconcile conflicting updates. The key design principle behind CRDTs is commutativity. The idea is that if the replicated state is only updated by commutative operations, then updates can be applied in any order and the replica states are still guaranteed to converge. In general, however, commutativity is not a common occurrence among data types. Most common data type definitions come with at least a pair of operations that do not commute. For instance an `add` and a `remove` operations on a set do not commute if both are for the same element. Likewise two `insert` operations for the same position in a list do not commute. To promote such a non-commutative data type to a CRDT, creative re-engineering of its internal representation and algorithms is needed. Many CRDTs have carefully-engineered implementations that keep track of various kinds of causal dependencies to help them identify and resolve conflicts. For instance, CRDTs implementing the set abstract data type rely on *vector clocks* to identify conflicting `add`s and `remove`s [11, 12], Replicated Growable Array (RGA) – a CRDT for collaborative editing [8], and JSON CRDT – a CRDT variant of the JSON storage format, both use *Lamport timestamps* [6, 8]; and TreeDoc, another CRDT for collaborative editing, makes use of *dense linear orders* [7]. Such advanced implementations makes it hard to reason about basic correctness properties of CRDTs, such as convergence. The case in point is the substantial effort and expertise required to verify *strong eventual consistency* of RGA, OR Set, and counter CRDT implementations [4]. The high cost of building CRDTs is a deterrent to their practical adoption. The key to overcome this deterrent is to let developers reuse their sequential abstractions in a distributed setting with little to no additional overhead of reasoning about their convergence properties.

In this paper we describe an alternative approach to building convergent replicated data types that realizes the aforementioned virtues. Our approach is based on Mergeable Replicated Data Types (MRDTs) [5] – an alternative take on RDTs that is inspired by the Git version control system. MRDTs adopt a state-centric model of replication based on version-controlled *mergeable* states instead of an operation-centric model based on commutative operations. Unlike commutativity, mergeability does not require a data type definition to be refactored to suit distributed execution. However, mergeability itself doesn't guarantee convergence. The key distinguishing characteristic of our approach is the extension of MRDTs with a distributed runtime that orchestrates only the *well-formed* convergent executions. Consequently, all MRDT executions are guaranteed to converge regardless of the chosen merge semantics. Notably, our runtime achieves this without impacting per-operation latency and

PaPoC, April 2022, Rennes, France
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

system-wide availability. Such a system is possible within the confines of the CAP theorem as it does not guarantee the linearizability of MRDT executions; only their convergence. Sec. 2 motivates our approach and demonstrates the underlying key intuitions Sec. 3 describes our runtime Quark with help of minimal formalism. We also describe our implementation of Quark atop Scylla – an off-the-shelf distributed data store [9]. Sec. 4 presents an evaluation case study that highlights the tradeoffs of our approach.

2 CRDTs to Convergent MRDTs

Asynchronous replication often leads to divergent executions as illustrated by Fig. 1 for `Set` abstract data type with `add` and `remove` operations.

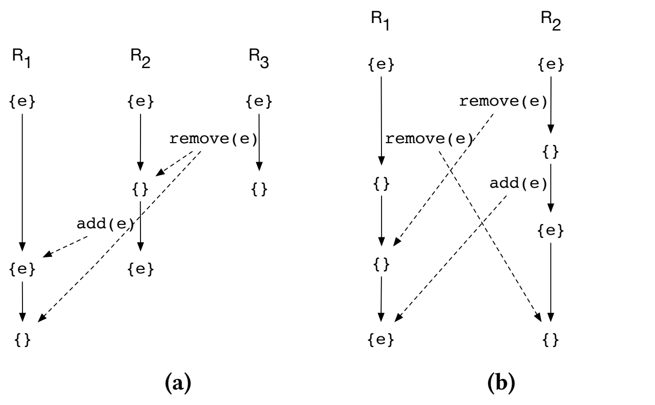

Figure 1. Anamolous executions resulting from the Asynchronous replication of `Set`. Dashed lines denote effect propagation.

Fig. 1a shows a divergent execution with three replicas – R_1, R_2, and R_3, all of which start with a singleton set containing the element e. A client connects to the replica R_3 and executes a `remove(`e`)` operation, which is then asynchronously propagated to other replicas. Some time after applying R_3's `remove` at R_2, another client connects to R_2 and re-adds e by issuing an `add(`e`)` operation. Consequently, the state at R_2 is again the singleton set $\{e\}$. Replica R_3 however receives R_2's `add` ahead of R_1's `remove`, applies them in the same order, and ends up with an empty set. The execution thus results in divergent replica states.

The CRDT Approach. It is clear that causal consistency would preempt the execution in Fig. 1a. The first step of building a `Set` CRDT is therefore enforcing causal consistency. This is done by extending the `Set` ADT implementation with vector clocks to keep track of the causal history of each operation, followed by applying the operations in the causal order at all replicas. This would eliminate the execution in Fig. 1a, but a different divergent execution as shown in Fig. 1b is still possible. Here, replicas R_1 and R_2 both start with a singleton set $\{e\}$. Two distinct clients connect to R_1 and R_2 respectively and issue two concurrent `remove(`e`)` operations. Later, another client connects to R_2 and issues an `add(`e`)` operation. The effects of these operations are asynchronously applied at remote replicas as shown in the figure, resulting in the divergent states at R_1 and R_2.

Note that, unlike in the previous execution, the conflicting operations here are not causally related. CRDTs require an *arbitration order* to be defined to order such concurrent conflicting operations. For example, in Fig. 1b, one might order R_2's `add` after R_1's `remove` letting the `add` operation *win*. Enforcing arbitration order however requires tracking element-wise causal dependencies between the `remove`s and `add`s by maintaining a vector clock *for each element* e in the set [11]. A `remove` operation on element e is applied at a replica R only if R hasn't seen a concurrent or later `add` operation on e (as determined by e's vector clock). The resultant *add-wins* set CRDT pre-preempts diverging executions of Fig. 1. However, the infrastructure needed to track and enforce causal and arbitration orders significantly complicates its implementation.

The MRDT Approach. Mergeable Replicated Data Types (MRDTs) implement a state-centric model of replication that is inspired by the Git version control system. Like in Git, the state evolves in terms of versions, and concurrent versions of the state can be merged. The semantics of merge depends on the type of the state, so each MRDT is required to be equipped with a three-way `merge` function that merges concurrent versions of that type in presence of their (lowest) common ancestor version. In our running example, the state is a value of type `Set.t`, hence `Set.merge` function would have the type signature:

$$\texttt{Set.merge} : \texttt{Set.t} \longrightarrow \texttt{Set.t} \longrightarrow \texttt{Set.t} \longrightarrow \texttt{Set.t}$$

The three arguments of `merge` correspond to the lowest common ancestor (LCA) version and the two concurrent versions that independently evolved from the LCA version. The LCA is a causal ancestor of concurrent versions, hence causal consistency is built into the replication model. The result of set merge, intuitively, must contain the common elements in the two concurrent versions along with any newly added elements in either versions. Concretely:

```
let merge s(*lca*) s1 s2 =
          (s1 ∩ s2) ∪ (s1 - s) ∪ (s2 - s)
```

Extending `Set` ADT with the above merge function results in a `Set` MRDT.

Let us now reconsider the executions in Fig. 1, this time on `Set` MRDT. The equivalent executions are shown in Fig. 2. Due to causal consistency being built into the model, the divergent execution of Fig. 1a is preempted in favor of the convergent execution shown in Fig. 2a. The execution manifests as follows. The initial version (v_0) on all three replicas is the singleton set $\{e\}$. Applying operations to replicas creates new versions, e.g., v_1 on R_3, and v_2 and v_3 on R_2. Changes can

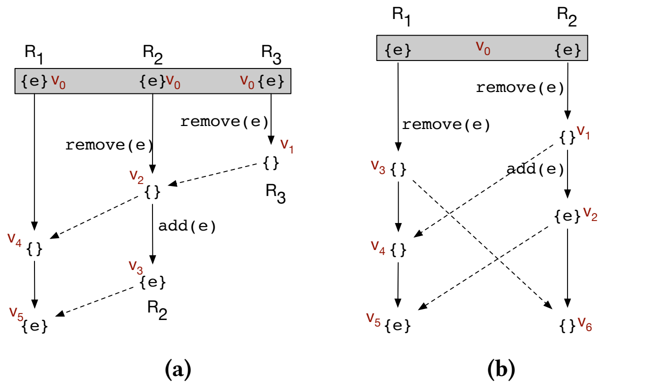

Figure 2. Equivalent executions of Fig. 1 in state-centric replication model. Dashed lines now denote state merges.

be propagated by merging versions, e.g., version v_2 on R_2 is a result of merging v_1 into v_0 for which v_0 serves as the lowest common ancestor (LCA) [1]. Likewise v_4 on R_1 is created by merging v_2 into v_0 in presence of their LCA v_0. The next version v_5 on R_1 is the result of merging R_2's v_3 into R_1's v_4. The LCA for this merge is v_2. By the end of the execution, versions v_5 and v_3 on R_1 and R_2 (resp.) have witnessed the same set of operations, hence are in agreement.

Unfortunately, the MRDT execution in Fig. 2b still diverges. Here R_1 and R_2 start with version $v_0 = \{e\}$. R_2 performs a remove and and an add making versions v_1 and v_2 respectively. Simultaneously, R_1 performs a remove to make v_3. Replica R_1 now obtains R_2's changes by merging versions v_1 and v_2 to make new versions v_4 and v_5 respectively. LCAs for these merges are v_0 and v_1. Concurrently, R_2 obtains R_1's changes by merging v_3 with v_2 (LCA = v_0) to make v_6. Now R_1 and R_2 have same set of changes yet their final versions differ.

Linearizing Merges. Note that the anamolous execution in Fig. 2b could have been avoided had the merges between R_1 and R_2 been linearized. Fig. 3a shows an execution that only slightly differs from the one in Fig. 2b. The difference is that the merges in Fig. 3a happen linearly: first R_1 is merged into R_2 (bringing R_1's remove), then R_2 into R_1 (bringing R_2's remove), followed again by R_2 into R_1 (bringing R_2's add). As a result of such linearization, the final versions on R_1 and R_2 converge to the singleton set $\{e\}$. Note that there exist other linearizations of merges; for instance, $R_1 \longrightarrow R_2$ merge could be ordered between the two $R_2 \longrightarrow R_1$ merges. However, all linearizations result in the same final state $\{e\}$. Another important point to note is that only the merges are linearized; not the entire execution. In Fig. 3a, both R_1 and R_2's removes remove the same element e, which is not possible in a linearized execution. Leaving the execution

[1]Versions v_0 and v_1 are not concurrent as the former is an ancestor of the latter. Merging v_1 into v_0 is nonetheless possible as v_1 is ahead of v_0 in causal order. In Git parlance this is a *fast forward* merge.

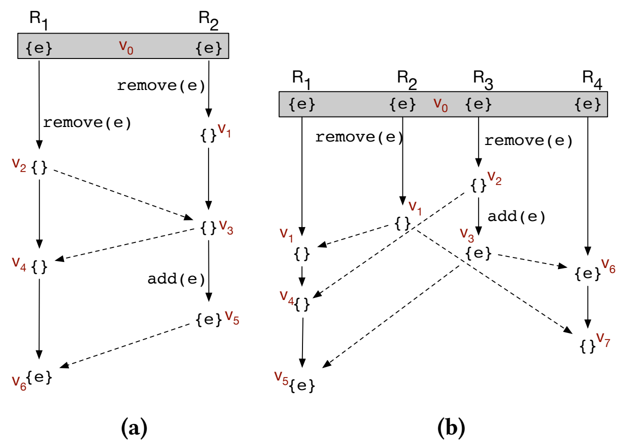

Figure 3. Executions demonstrating the difference between linearized (former) and concurrent (latter) merges.

unconstrained is crucial to satisfy the limits imposed by the CAP theorem on a highly-available partition-tolerant distributed system.

In the context of Fig. 3a, it is quite clear what linearization of merges means and how to enforce it via synchronization (for e.g., wrapping each merge within a global lock). In general however, the semantics of merge linearization isn't as cut and dried. For instance, consider the execution in Fig. 3b. The four replicas involved in the execution start with version $v_0 = \{e\}$. The replicas perform local operations as shown in the figure to make versions v_1 to v_3. Next they perform a series of merges to propagate local changes. The merges can be ordered in time as following: first $R_2 \longrightarrow R_1$ (merging v_1), then $R_3 \longrightarrow R_1$ twice (merging v_2 and v_3 resp.), then $R_3 \longrightarrow R_4$ (merging v_3), and finally $R_2 \longrightarrow R_4$ (merging v_1). These merges collectively propagate the effects of add and remove operations to R_1 and R_2. And despite being ordered in time, the they nonetheless result in a divergent execution ($v_5 \neq v_7$). The problem here is that, although merges are executed linearly, the execution graph does not reflect this linearity; merges of that end in R_1 in Fig. 3b are effectively concurrent with those that end in R_4. This shows that simply synchronizing the execution of merges does not necessarily result in convergence.

Our key insight to overcome this impasse is a *well-formedness* condition on execution graphs that ensures convergence of final states. To understand well-formedness, let us contrast the bad executions in Figs. 2b and 3b against the good execution in Fig. 3a. Observe that in Fig. 3a, *every* pair of concurrent versions on R_1 and R_2 have a unique lowest common ancestor (LCA). For instance, LCA of (v_5, v_6) is v_5, (v_5, v_4) is v_3, (v_1, v_2) is v_0 and so on. By contrast in Fig. 2b, versions v_5 and v_6 have two LCAs, namely v_2 and v_3. Both these versions are common ancestors of v_5 and v_6, and both are *lowest* in the sense that there do not exist versions lower (in the execution graph) than v_2 and v_3 that are also common ancestors of v_5 and v_6. Likewise in Fig. 3b, versions v_7 and v_5 have two LCAs – v_1 and v_3. Multiple LCAs is an indication that there exist merges prior in the execution graph that are effectively concurrent. Considering that our approach to convergence in state-centric model crucially relies on linearity of merges, presence of multiple LCAs opens up the possibility of divergence. We therefore define well-formed execution graphs as those where LCAs for every pair of concurrent versions is unique. We have formally proved that enforcing this structural well-formedness condition indeed guarantees the convergence of distributed executions. The next section covers how this condition is efficiently implemented and enforced in a distributed runtime we built called Quark. Such automatic enforcement of convergence lets us promote ordinary sequential data types to convergent replicated types by simply equipping them with a merge function.

3 Quark

The aim of Quark runtime is to manifest only the well-formed distributed executions of MRDTs. As described in Sec. 2, well-formed executions are precisely those where every pair of versions have a unique lowest common ancestor (LCA). The violation of unique LCA property can only happen during a merge operation as illustrated by Figs. 2b and 3b. Quark enforces unique LCA property at run-time by allowing only the *safe* merges that do not result in multiple LCAs. To demonstrate this intuition, we embrace the analogy with Git, and envision a distributed execution as a process that progressively builds a version history graph $G = (V, E)$. Vertices of this graph are versions and edges ($\longrightarrow$) denote causal relationships between versions. For versions v_0 and v_1, $v_0 \longrightarrow^* v_1$ means that there is a path from v_0 to v_1 in the version history graph, which in-turn means that v_0 causally precedes v_1. A branch b is a linear sequence of edges that denote the evolution of state at a replica R_b. In other words, there is a one-to-one correspondence between branches and replicas. We let $H(b)$ denote the "head" of a branch b, i.e., the latest version on replica R_b. For simplicity in the formal model, we only allow latest versions on replicas (i.e., the branch heads) to be merged. Let $L(v, v')$ denote the LCA of versions v and v'. We trivially lift the notion of LCA from versions to branches by defining the LCA of branches b and b', denoted $L(b, b')$, to be the LCA of their heads, i.e., $L(H(b), H(b'))$. With these definitions in place, we can now explain how unique LCA property can be enforced.

Fig. 4 captures the scenario of merging (the head of) a branch b' into b. Let b'' be another branch. We consider the most general case when (i). Branches b, b', and b'' are distinct, and (ii). Their LCAs $L(b, b'')$ and $L(b', b'')$ lie on a distinct pair of branches not equal to b, b', and b''. In the figure, once you merge b' into b, every version v that is an ancestor of $L(b', b'')$, i.e., $v \longrightarrow^* L(b', b'')$, will be a common ancestor of

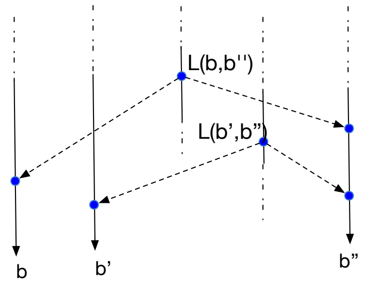

Figure 4. Merge of branch b' into b is considered safe iff for every other branch b'', it is the case that $L(b, b'') \longrightarrow^* L(b', b'') \lor L(b', b'') \longrightarrow^* L(b, b'')$.

$H(b)$ and $H(b'')$. Clearly, $L(b', b'')$ is the lowest among such common ancestors. But the current lowest common ancestor of b and b'' is $L(b, b'')$. We therefore end up with two lowest common ancestors – $L(b', b'')$ and $L(b, b'')$, *unless* both are ancestrally related. It follows that for this merge to be safe it must be the case that either $L(b, b'') \longrightarrow^* L(b', b'')$, or $L(b', b'') \longrightarrow^* L(b, b'')$. Quark checks for this condition each time a merge is attempted during the execution. It allows the merge only if the condition is satisfied, thereby enforcing the unique LCA property, and in-turn convergence.

Garbage Collection. One downside of the aforementioned approach is that the version history grow monotonically as the execution progresses and new versions are created. Fortunately, this is easy to address as the execution only needs finite version history to compute LCAs. Our implementation assigns a version vector to each unique version. The version vector of the result of a merge operation is computed as the least upper bound ($\sqcup$) of merging versions. Conversely, the version vector of their LCA is computed as their greatest lower bound ($\sqcap$). We prove that LCA version vectors monotonically increase, which means that older versions with vectors less than the least known LCA vector can simply be garbage-collected. Moreover, a replica can make this decision locally without having to synchronize with its peers. In practice, applications may prefer to flush out older version history to a stable storage from where it can be re-created as necessary.

Implementation. We implemented a prototype of Quark runtime as a lightweight shim layer on top of Scylla – an off-the-shelf distributed data store [9]. We rely on Scylla for inter-replica communication, data replication, persistence, and fault tolerance. Quark translates the high-level mergeable data type implementations in OCaml to their low-level representations in the backing store and organizes their well-formed distributed executions. The synchronization needed to linearize merges is implemented with help of Scylla's support for conditional updates (CAS operations) and expiring columns. The total order among merges is enforced with help of Quorum reads and writes. Each user process is assigned its own replica of the RDT. Version vectors are realized as associative lists and stored in Scylla as blobs.

4 Evaluation

We present an evaluation study of our runtime-assisted approach to convergent RDTs with help of our prototype Quark implementation. The benchmark we chose for the study is a collaborative document editing application – a common usecase addressed by several CRDT proposals [7, 8, 10]. However, unlike these other approaches, we did not have to build a dedicated replicated data type to represent collaboratively-edited documents; an ordinary document format extended with a merge operation would suffice. While many data structures exist to represent text documents (e.g., ropes [1]), we decided to adopt the simplest representation of a document as a list of characters.

```
type doc = char list
```

While being simple, the advantage of this presentation is that we can simply reuse the three-way List.merge function of the list data type to merge documents. List.merge is a simple implementation of the GNU diff3 algorithm [3] in 60 lines of OCaml. We thus adopt a straightforward approach to building a collaborative document editor with the intention to keep the development effort low enough to be easily replicated. The convergence guarantee of Quark ensures that the simplicity of our implementation doesn't come at the expense of correctness. We now describe an evaluation that demonstrates that it also doesn't come at the expense of the editor performance.

Our experiment setup consists of multiple collaborators simultaneously editing a 10000+ line document obtained from the Canterbury Corpus [2]. Each user holds a replica of the document and is assumed to be editing the document at the speed of 240 characters per minute or 1 character every 0.25s. At 6 characters per word, this amounts to 40 words per minute, which is the average typing speed of humans. Each edit is immediately persisted to the disk by creating a new version in the backing store. Thus there are at least as many versions of the document as there are edits. Such extensive versioning may be considered excessive in practice and could be disabled. Each user process runs a Quark thread that commits user-generated document versions to the local branch, while merging with the concurrent versions from the remote branches in the background. Each merge is synchronized as described in previous sections.

Quark's background merges however pose a new problem as they create new versions on the local branch in the background while the user is busy editing an older version. When the user attempts to write their version of the document to the store, simply committing it would effectively override the concurrent updates from other users obtained via background merges. The solution, fortunately, is straightforward: we *merge* the user-submitted value with the (value of) the latest version on the local branch to create a new version that includes the updates from either direction. Well-formedness check is not needed for this merge as it is guaranteed by construction. Thus, Quark's write is a function of type Value → Value, where Value is doc in the current application.

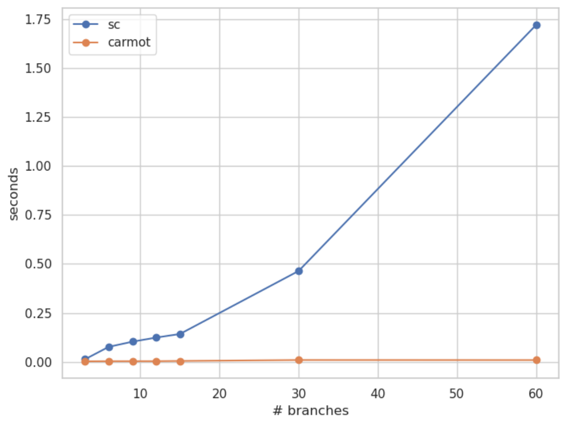

Figure 5. Latency of collaborative editing operations under Quark vs executing them with strong consistency (SC).

To measure the impact of Quark runtime on user writes, we measure the latency of the write operation, which includes for the time spent merging the user version with the current version, and persisting the resultant version to the store. We conduct the experiments on a three-node cluster of i3.large machines in Amazon us-west2 data center. Each user connects to one of the machines and performs 1000 edits in succession, saving the document after each edit. We progressively increase the number of concurrent users editing the document from 3 to 60 and measure the impact of the increased concurrency on write latency. Fig. 5 shows the median latency values. As evident from the figure, latency of Quark writes remain more-or-less constant with the median latency around 0.008s. The maximum latency ever measured for a Quark writes is 0.016s. Notably, latency remains an order of magnitude less than the time between consecutive edits (0.25s), making it hard to perceive. Fig. 5 also plots the latency values for the baseline "SC" approach which achieves convergence by synchronizing each operation, i.e., executing under strong consistency (SC). The SC write latency increases super-linearly with the median value of 1.76s for 60 concurrent editors. The maximum latency measured for an SC write is 9.02s, which is considerably more than the inter-edit latency of 0.25s. The experiment confirms the theory that convergence does not come at the cost of latency in Quark.

Quark system replicates the contents of each branch across all the replicas as fast as the network allows. However,

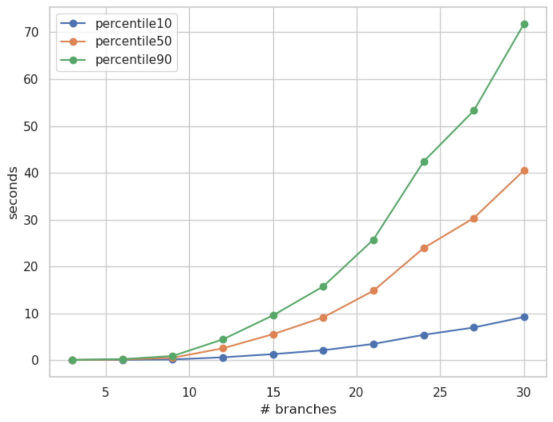

Figure 6. Staleness increases as the number of concurrent editors increase.

for a user A to see the changes made by the other user B, the changes have to be reflected in A's local version, which can only happen through a merge operation. Since Quark synchronizes merge operations globally, it induces additional delay before A can see B's changes. We call this additional delay *staleness* as with the progression of time, B's version known to A becomes increasingly stale. At the system-level, an increase in staleness effectively delays the convergence (but doesn't preempt it). To understand the staleness behavior in Quark, we repeat the collaborative editing experiment, this time measuring the staleness value at every merge. We do this by annotating every version v with the timestamp t of its creation time. When v is is merged into a remote branch b at a later time t', the difference $t' - t$ denotes the staleness of v w.r.t the new version on b. Multiple such staleness measurements are recorded for each experiment to compute the 10th, 50th, and 90th percentile values. Fig. 6 shows the results. While staleness remains in the order of milliseconds with fewer (≤ 9) concurrent users, it increases super-linearly as we increase the number of collaborators in the multiples of 3 until 30. While increased staleness is inevitable in our approach due to synchronized merges, we believe the increase can be contained by choosing the order of merges to avoid the "starvation" of some branches. Note that relaxing the linearizability constraint on merges would completely eliminate the staleness overhead, but the resultant Git-like system fails to converge due to anomalous executions described in Sec. 2[2].

Our experiments bring to the fore an inherent tradeoff among the competing concerns of RDTs, namely (i). The ease of programming convergence, (ii) Latency, and (iii). Staleness. While CRDTs try to optimize for latency and staleness, they require a significant amount of development and verification effort to be expended to ensure convergence [4]. In contrast, Quark lets developers derive convergent-by-construction RDTs from ordinary data data types that are optimized for latency, but incur a staleness overhead that delays the time to convergence.

[2]Git admits anamolous version history graphs where two branches can have the same set of commits and yet differ in their final version. Supplementary material describes two such cases we observed on Github.

References

[1] Hans-J. Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: An Alternative to Strings. *Softw. Pract. Exper.* 25, 12 (Dec. 1995), 1315–1330. https://doi.org/10.1002/spe.4380251203

[2] Canterbury 2021. The Canterbury Corpus. https://corpus.canterbury.ac.nz/descriptions/ Accessed: 2021-11-18 13:21:00.

[3] GNU Diffutils 2021. GNU Diffutils. https://www.gnu.org/software/diffutils/ Accessed: 2021-11-18 13:21:00.

[4] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 109 (oct 2017), 28 pages. https://doi.org/10.1145/3133933

[5] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (oct 2019), 29 pages. https://doi.org/10.1145/3360580

[6] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 2733–2746. https://doi.org/10.1109/tpds.2017.2697382

[7] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS '09)*. IEEE Computer Society, Washington, DC, USA, 395–403. https://doi.org/10.1109/ICDCS.2009.20

[8] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.* 71, 3 (mar 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[9] Scylla 2021. A Real-Time Big Data Database. https://www.scylladb.com/ Accessed: 2021-11-18 13:21:00.

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS'11)*. Springer-Verlag, Berlin, Heidelberg, 386–400. http://dl.acm.org/citation.cfm?id=2050613.2050642

[11] Marek Zawirski. 2015. *Dependable Eventual Consistency with Replicated Data Types.* Ph.D. Dissertation.

[12] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Jian Lu. 2019. Remove-Win: a Design Framework for Conflict-free Replicated Data Collections. arXiv:1905.01403 [cs.DC]