Extending JSON CRDTs with Move Operations

Liangrun Da
liangrun.da@tum.de
Technical University of Munich
Munich, Germany

Martin Kleppmann
martin@kleppmann.com
University of Cambridge
Cambridge, UK

Abstract

Conflict-Free Replicated Data Types (CRDTs) for JSON allow users to concurrently update a JSON document and automatically merge the updates into a consistent state. Moving a subtree in a map or reordering elements in a list within a JSON CRDT is challenging: naive merge algorithms may introduce unexpected results such as duplicates or cycles. In this paper, we introduce an algorithm for move operations in a JSON CRDT that handles the interaction with concurrent non-move operations, and uses novel optimisations to improve performance. We plan to integrate this algorithm into the Automerge CRDT library.

CCS Concepts: • **Theory of computation** → **Distributed algorithms**; • **Software and its engineering** → **Consistency**; • **Information systems** → *Collaborative and social computing systems and tools.*

Keywords: conflict-free replicated data types, replica consistency, JSON, tree data structures, move operation

ACM Reference Format:
Liangrun Da and Martin Kleppmann. 2024. Extending JSON CRDTs with Move Operations. In *The 11th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642976.3653030

1 Introduction

Automerge [2] is an implementation of Conflict-Free Replicated Data Type (CRDT) [15], which allows concurrent changes to data on different devices to be merged automatically without requiring any central server. It is used in the development of local-first software [7], which includes applications such as collaborative drawing, text editing, and more.

Automerge uses a *document* as its data model, which can be viewed as a JSON data type. A document can be accessed locally through operations such as *get*, *put*, and *delete*, and any modifications are replicated to other devices. Many applications require reordering elements in a list, e.g. when using drag-and-drop to change the order of a to-do list. Moreover, some applications need to move a subtree to a new parent node: for example, in a document representing a file system tree, moving a file or directory from one location to another is a very common operation.

Although move operations are straightforward for an unreplicated JSON object, as they only require deletion and reinsertion, they become challenging in the context of CRDTs. If several replicas concurrently delete and reinsert the same object, the merged result contains duplicates of the moved object [4]. Another issue arises with cycles, as illustrated in Figure 1. In this example, each device individually executes a move operation without creating a cycle. However, when merging these operations, a cycle may appear if the algorithm does not take care to prevent the cycle. Currently, Automerge handles moves by deletion and reinsertion, which results in behaviour (b) in Figure 1.

Previous research has demonstrated the possibility of move operations on lists and trees [4, 6, 9, 10], but the existing algorithms treat the children of a tree node as an unordered set, rendering them unsuitable for direct application to JSON trees. In JSON trees, a branch node is either a key-value map or an ordered list. Therefore, the CRDT algorithm must handle operations on those maps and lists that may interact with concurrent move operations (for example, by overwriting a key containing a node being moved). Moreover, the algorithm needs to be able to move elements between a list and a map. In this paper, we show how to implement a move operation with this functionality. We also developed novel optimisations to improve its performance. We implemented the algorithm as a standalone prototype to facilitate experimentation with various algorithm variants, showcasing the impact of these optimisations and the practical feasibility.

2 The Core Mechanics of Automerge

Automerge uses a monotonically growing set of *operations* called *OpSet* as its internal representation. The set of operations can be exchanged between different peers, and two sets can be merged by taking their union. Figure 2 shows an example JSON document and its internal OpSet.

When modifying a document, a new operation is added to the OpSet. Operations are never removed from the OpSet. An operation sets or deletes the value on a single key, or a single element in a list.

Every operation in Automerge is assigned a unique ID, its *opID*, which is implemented as a Lamport Clock [8]. Every

PaPoC '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0544-1/24/04
https://doi.org/10.1145/3642976.3653030

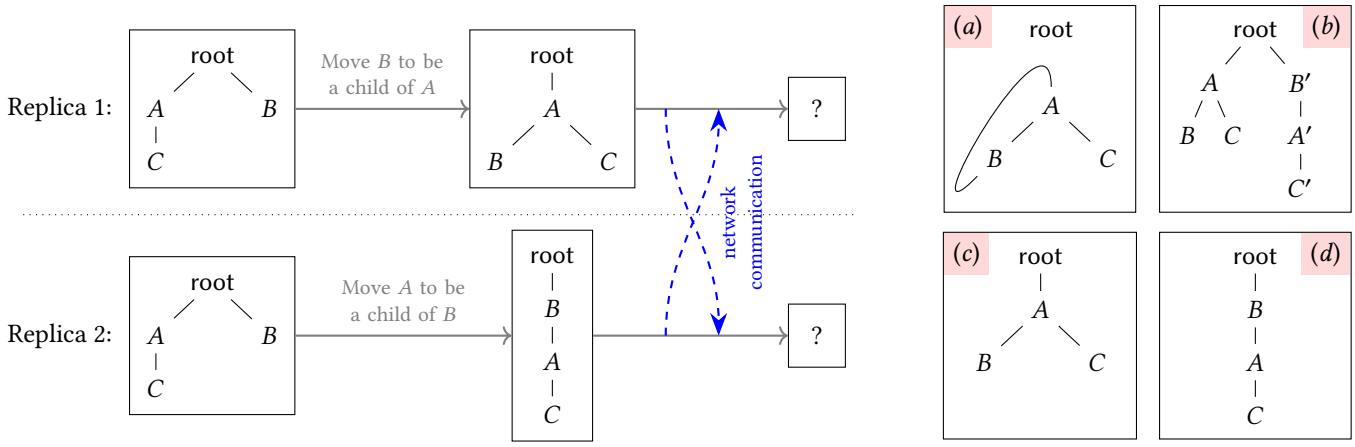
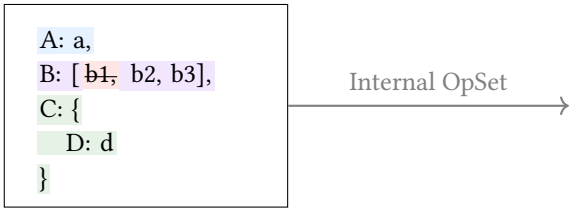


Figure 1. Initially, nodes A and B are siblings. Replica 1 moves B to be a child of A, while concurrently replica 2 moves A to be a child of B. Boxes (a) to (d) show possible outcomes after the replicas have communicated and merged their states: (a) A and B form a cycle; (b) concurrently moved subtrees are duplicated; (c) Replica 2's move is ignored; (d) Replica 1's move is ignored. Figure from [6].



ID	Type	Object ID	Key	Value	Predecessors
⟨1, 0⟩	put	⟨0, 0⟩	A	a	
⟨2, 0⟩	make	⟨0, 0⟩	B	list	
⟨6, 0⟩	make	⟨0, 0⟩	C	map	
⟨7, 0⟩	put	⟨6, 0⟩	D	d	
⟨3, 0⟩	put	⟨2, 0⟩	⟨0, 0⟩	b1	
⟨4, 0⟩	put	⟨2, 0⟩	⟨3, 0⟩	b2	
⟨5, 0⟩	put	⟨2, 0⟩	⟨4, 0⟩	b3	
⟨8, 0⟩	delete				⟨3, 0⟩

Figure 2. An example JSON document with its internal OpSet

operation references the ID of the list or map object that it modifies, which is stored as the *object ID* of the operation. A list or map is created using a "make" operation, and the ID of this operation subsequently serves as the unique identifier for the object it created.

If an operation op_2 overwrites, deletes or moves an existing key in a map or element in a list, and that existing value was assigned by a prior operation op_1, we say that op_1 is a *predecessor* of op_2, and op_2 is a *successor* of op_1. Every operation includes the opIDs of its immediate predecessors. Multiple predecessors could exist, as several prior operations may concurrently assign values to the same key. An operation is invisible if it has one or more successors.

3 The Moving Algorithm

When the user moves an element from one location to another, a move operation is generated and added to the OpSet. This move operation identifies the element being moved and where it is moved to. The destination of the move is determined by an object ID and a key if the destination object is a map, or a list element ID if it is a list. The ID of a prior put or make operation identifies the element being moved, and a move operation has a *MoveID* field where this ID is stored.

3.1 Validity of Move Operations

If multiple concurrent move operations move the same element, we define the move operation with the largest ID as the winner, and others have no effect. This prevents the concurrent moves from duplicating the element. A move operation is defined to be valid if and only if there is no concurrent move operation with a greater ID that moves the same element, and it does not introduce any cycles. If a move operation is invalid, the move operation itself is invisible and its predecessors remain visible (unless they have another successor operation that is valid).

The challenge is to ensure a consistent decision among replicas regarding the validity of the same operation. Consider Figure 1, where Replica 1 moves B to be a child of A and then receives the change from Replica 2, while Replica 2 moves A to be a child of B and then receives the change from Replica 1. If we check the validity at the time of applying operations, Replica 1 would consider the operation of

Algorithm 1 A naive approach for updating validity of operations

```
Input: ops - operations in the OpSet in ascending ID order
Output: valid - a map from operation ID to its validity
 1: procedure UpdateValidity(ops)
 2:     tree ← {}          ▷ a map from child ID to parent ID
 3:     winners ← {} ▷ a map from object ID to winner ID
 4:     valid ← {}
 5:     for op in ops do          ▷ in order of ascending op.ID
 6:         for pred in op.Predecessors do
 7:             tp ← pred.Type
 8:             if tp == move and valid[pred.ID] then
 9:                 tree[pred.MoveID] ← null
10:             else if tp == make then
11:                 tree[pred.ID] ← null
12:         if op.Type == make then
13:             tree[op.ID] ← op.ObjectID
14:         else if op.Type == move then
15:             mid ← op.MoveID
16:             oid ← op.ObjectID
17:             if IsAncestor(tree, oid, mid) then
18:                 valid[op.ID] ← false
19:                 continue
20:             valid[op.ID] ← true
21:             tree[mid] ← oid
22:             prevWinner ← winners[mid]
23:             if prevWinner != null then
24:                 valid[prevWinner] ← false
25:             winners[mid] ← op.ID
26:     return valid
27:
28: function IsAncestor(tree, node, ancestor)
29:     while true do
30:         if node == ancestor then
31:             return true
32:         if node == null || node == root then
33:             return false
34:         node ← tree[node]
```

moving A to be a child of B as invalid, while Replica 2 would say the operation of moving B to be a child of A is invalid, resulting in inconsistent document states on two replicas.

To ensure consistent decisions on the validity of operations, we apply operations in ascending ID order. We first show a simple but inefficient approach in Algorithm 1: whenever an operation is inserted into OpSet, all the operations in the OpSet are reapplied, and the validity of operations is updated accordingly. We present an optimized algorithm in Section 3.2.

We maintain a map named *tree* to keep track of the parent-child relationship between objects. All objects within a list are considered children of the list object, and the same applies to map objects. When an object is deleted from the tree, we record this in the map by setting the deleted object's parent to *null*. A deletion can be thought of as similar to moving the deleted object to a "trash" tree that is separate from the visible document tree.

Additionally, we maintain another map named *winners*, whose key is the ID of an element being moved, and the associated value is the greatest ID among operations that move this element.

By reapplying the operations, we update the tree to reflect any changes in the parent-child relationship in the document. Lines 6 to 11 set the parent to null for any objects that are deleted or overwritten. Lines 14 to 25 update the validity by checking for cycles and concurrent moves.

In Figure 1, the result (d) implies replica 2's operation precedes replica 1's. From replica 1's perspective, its operation starts as visible but becomes invisible after receiving the operation from replica 2. It's possible to construct more complex scenarios in which an operation goes from invisible to visible as a result of receiving a remote operation.

3.2 Performance Optimisation

Algorithm 1 ensures the consistency of move operations by applying them in the same order on all replicas. However, reapplying the entire set of operations in the OpSet is very inefficient.

To optimise the algorithm, we can avoid executing the parts that repeat the previous invocation of *UpdateValidity*. Let n be the ID of the newly added operation. In Algorithm 1, the execution of the loop for operations with an ID less than n is the same as the previous invocation. During real-time collaboration, the operations with IDs greater than n are usually a small fraction of the total set of operations.

Algorithm 2 shows an optimised version of Algorithm 1. To insert a new operation with ID n into OpSet, our algorithm first restores the parent-child relationship to its state when only operations with IDs less than n are applied. We then apply the new operation, followed by reapplying the operations with IDs greater than n. We call this sequence of steps the *Restore-Apply-Reapply (RAR)* procedure.

To restore the parent-child relationship to a past state, we store former parents of overwritten, deleted, and moved objects in the variable *parents*, in which we maintain a map from object IDs to the old parent IDs for each operation.

Additionally, to restore the validity status of operations with a lower ID, we maintain a stack of move operations for every object that is moved. Operations in a stack are sorted

Algorithm 2 The Restore-Apply-Reapply approach for updating validity of operations

```
1:  procedure UpdateValidity
2:      ops ← []                                                    ▷ operations in ascending ID order
3:      tree ← {}                                                   ▷ a map from child ID to its parent ID
4:      moves ← {}          ▷ a map from element ID to a stack of move operation IDs for moving this element
5:      parents ← []                ▷ a list, where each element is a map, from object ID to its parent ID
6:      valid ← {}
7:      while whenever the local replica receives an operation o do
8:          Insert o into ops at index i such that all operations at indexes > i have an ID greater than o.ID, and all operations at indexes < i have an ID less than o.ID
9:          Insert a new map into parents at index i
10:         for k ← |ops| - 1 to i + 1 do                                                    ▷ Restore
11:             for (object, location) in parents[k] do
12:                 tree[object] = location
13:             if ops[k].Type is move and valid[ops[k].ID] then
14:                 moves[ops[k].MoveID].pop()
15:                 prevMove ← moves[ops[k].MoveID].peek()
16:                 if prevMove != null then:
17:                     valid[prevMove] ← true
18:         for k ← i to |ops| - 1 do                                                    ▷ Apply and Reapply
19:             op ← ops[k]
20:             parent ← {}
21:             prevParent ← null
22:             if op.Type is make then
23:                 tree[op.ID] ← op.ObjectID
24:                 parent[op.ID] ← null          ▷ restore a make operation is to delete the object it creates
25:             else if op.Type is move then
26:                 if op.Value == null then                ▷ the value is not null if it moves a scalar value
27:                     if IsAncestor(tree, op.ObjectID, op.MoveID) then
28:                         valid[op.ID] ← false
29:                         continue
30:                     prevParent = tree[op.MoveID]
31:                     tree[op.MoveID] ← op.ObjectID
32:                 valid[op.ID] ← true
33:                 if moves[op.MoveID] == null then
34:                     moves[op.MoveID] ← new stack
35:                 prevMove ← moves[op.MoveID].peek()
36:                 if prevMove != null then:
37:                     valid[prevMove] ← false
38:                 moves[op.MoveID].push(op.ID)
39:             for pred in op.Predecessors do                  ▷ handle deleted and overwritten objects
40:                 if pred.Type is move and valid[pred.ID] then
41:                     parent[pred.MoveID] ← tree[pred.MoveID]
42:                     tree[pred.MoveID] ← null
43:                 else if pred.Type == make then
44:                     parent[pred.ID] ← tree[pred.ID]
45:                     tree[pred.ID] ← null
46:             if prevParent != null then
47:                 parent[op.MoveID] ← prevParent
48:             parents[k] ← parent
```

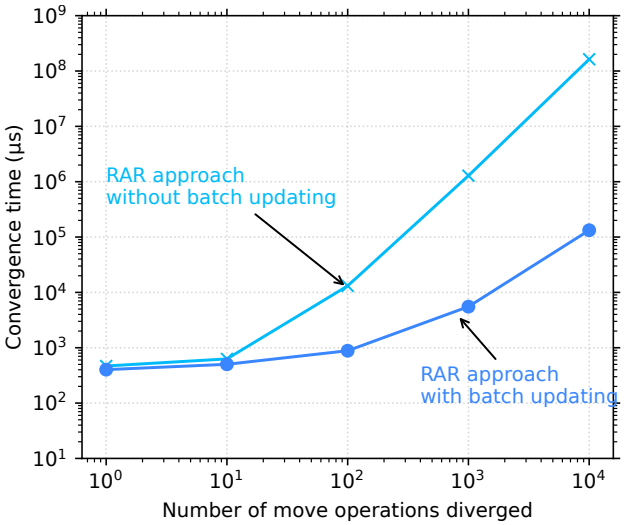


Figure 3. Convergence time of two actors that diverge by move operations

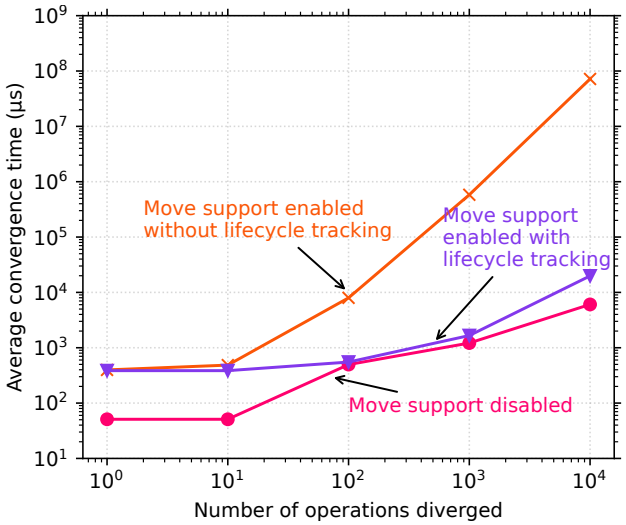


Figure 4. Convergence time of two actors that diverge by non-move operations

by ID, with the top of the stack being the winner among the concurrent move operations of the same element. To restore to a past state, all the move operations with IDs greater than that of the past operation are popped.

Lines 10 to 17 in Algorithm 2 restore the effects of the operations greater than the new operation. Lines 18 to 48 update the tree to reflect any changes in the parent-child relationship by applying the newly inserted operation and reapplying the following operations. The validity is updated by detecting cycles and concurrent move operations during applying and reapplying operations similarly to Algorithm 1. Additionally, we store the old parent of objects, ensuring the ability to restore to any past state. The algorithm doesn't check for cycles if the move operation moves a scalar value, as it is impossible to create a cycle in this case.

3.3 Further Optimisations

3.3.1 Batch Updating. Sometimes a large number of operations are applied at once, e.g. when a user has been working offline and comes back online. Instead of calling UpdateValidity for each operation, it is more efficient to collectively apply all operations at once, and thereby amortise the cost of restoring and reapplying.

3.3.2 Lifecycle Tracking. In the process of Restore-Apply-Reapply, objects shift between the tree and the trash as they are moved, deleted or overwritten. As each operation's ID is a logical timestamp, we can create a sequence of IDs for each object that traces these shifts over time. We call this sequence of IDs the *LifecycleList* of an object. The LifecycleList can be divided into two sublists: the PresentList, which consists of operations that create or move the object, and the TrashList, which also contains operations that overwrite or delete the object. Upon receiving an operation, we insert the ID of the new operation into the TrashList of objects it deletes or overwrites, as well as the PresentList of objects it creates or moves.

By storing the lifecycle of each object, it is no longer necessary to restore and reapply non-move operations, which merely shift objects back and forth between the tree and the trash, without changing their parents. Having to perform RAR only for move operations results in a large performance improvement for workloads where most operations create/delete objects and update values within objects, and only a small fraction of operations are moves.

4 Evaluation

We implemented the move algorithm in Go in a standalone prototype, which is a simplified version of Automerge. The source code is available on GitHub[1]. We plan to integrate the algorithm into the Rust implementation of Automerge [2] in the future. We ran the experiments on AWS c5.large instances, each having 4vCPU and 2GB RAM.

4.1 Convergence Complexity and Performance

To measure the time it takes for two divergent actors to converge, two actors start with identical documents containing 100 map objects. They each generate N move operations concurrently, and then send the operations to each other. For each move we choose a random object to be moved, and a random object as the destination.

Figure 3 shows the result of the experiment. With batch updating, when N = 100, it takes about 1 ms to converge, which is acceptable for most real-time collaborative applications.

[1] https://github.com/LiangrunDa/AutomergeWithMove

4.2 Overhead Caused by Move Support

Even when an application does not use any move operations, our algorithm needs to perform additional work on operations that modify the parent-child relationship of objects by creating, deleting or overwriting objects, compared to an implementation that does not support moves. To quantify this overhead, we measure the time to achieve convergence on our implementation with support for move operations enabled, and compare it to a version of the same implementation with support for move operations disabled by removing calls to UpdateValidity. Initially, both actors have identical documents containing 100 map objects. They each concurrently add a further N objects to the document, and then send the operations to each other. Batch updating is not used in this experiment.

Figure 4 shows the results. Without lifecycle tracking, the convergence time grows to more than 60 seconds for $N = 10^4$; with lifecycle tracking, this is reduced to 20ms, which is much closer to the 6ms it takes to converge with the move operation disabled.

4.3 Correctness Testing

We test the correctness of the implementation by randomly generating operations on different actors to check if they converge to the same state. This approach is inspired by Jepsen [3], a framework for distributed systems verification.

With this approach, we found several bugs during the design of the move algorithm. Those bugs were caused by not taking care of the corner cases with combinations of inputs that are rarely encountered during normal execution. Even if this approach cannot prove the correctness of the algorithm, it can uncover some subtle corner cases and provide confidence in its correctness.

5 Related Work

5.1 CRDTs for Trees

Several previous papers present designs for JSON CRDTs [1, 5, 14], but none of them support a move operation. On the other hand, there are several algorithms for move operations on trees, but they focus on managing the parent-child relationship, without integrating the map or list CRDT data types that occur in JSON:

- Previous work by Nair et al. [9] proposed a conflict-free replicated tree type with move operations. They categorized these moves into up-moves (towards the root) and down-moves (away from the root); concurrent up-moves are safe while cycles caused by other move operations are resolved by ignoring some operations.
- Najafzadeh et al. [10] proposed a fully asynchronous file system that replaces the move operations with copy-delete operations if a cycle occurs, leading to duplication of directories. They also proposed a mostly asynchronous file system that uses locks to coordinate concurrent move operations, but this approach is not available under network partitions.
- Kleppmann et al. [6] proposed a move operation for trees based on an undo-do-redo algorithm, which forms the basis of the algorithm in this paper. We change the name to Restore-Apply-Reapply to avoid confusion with a user-facing undo feature.

To our knowledge, our algorithm is the first to handle the combination of a move operation with a JSON tree structure, including the overwriting keys in map objects, moving multi-value registers within map and list objects, and reordering elements within list objects. These features make the algorithm significantly more complicated. Furthermore, we introduce the concept of lifecycle tracking to reduce the overhead of the algorithm, which is not considered in previous work on move operations.

5.2 CRDTs for Lists

There are many CRDTs for lists, such as Treedoc [13], WOOT [12], Logoot [16], and LSEQ [11]. However, none of them support move operations.

DSON [14] is a delta-based JSON CRDT that supports reordering of items in lists, but not tree moves that change the parent of a node.

Kleppmann introduced an algorithm to extend existing List CRDTs with move operations [4]. The algorithm uses an LWW register for each element to track the location of the element. We incorporate the algorithm into our Restore-Apply-Reapply procedure by tracking concurrent move operations that move the same element and selecting the move operation with the greatest ID as the winner.

6 Conclusions

In this paper, we present an implementation of move operations in a JSON CRDT, a tree of nested map and list objects. We optimise the procedure using batch updating, and the novel technique of lifecycle tracking.

Our performance experiments demonstrate the practical feasibility of the move operation, even in scenarios with large numbers of concurrent operations. We also evaluate the correctness of our implementation through randomised testing, ensuring that the algorithm converges to the same state across different actors. In future work we plan to integrate this algorithm into the Automerge CRDT library and formally verify its correctness.

Acknowledgments

This work was done while Martin Kleppmann was at the Technical University of Munich, funded by the Volkswagen Foundation and crowdfunding supporters including Mintter and SoftwareMill.

References

[1] Amos Brocco. 2022. Melda: A General Purpose Delta State JSON CRDT. In *9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2022)*. ACM, 1–7. https://doi.org/10.1145/3517209.3524039

[2] Automerge community. [n. d.]. Automerge CRDT. https://automerge.org. Accessed: 2023-07-26.

[3] Jepsen contributors. [n. d.]. Distributed Systems Safety Research. https://jepsen.io. Accessed: 2023-09-24.

[4] Martin Kleppmann. 2020. Moving elements in list CRDTs. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–6.

[5] Martin Kleppmann and Alastair R Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (April 2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382 arXiv:1608.03960

[6] Martin Kleppmann, Dominic P Mulligan, Victor BF Gomes, and Alastair R Beresford. 2021. A highly-available move operation for replicated trees. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2021), 1711–1724.

[7] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software(Onward!)*. 154–178.

[8] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[9] Sreeja Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, and Marc Shapiro. 2021. A coordination-free, convergent, and safe replicated tree. *arXiv preprint arXiv:2103.04828* (2021).

[10] Mahsa Najafzadeh, Marc Shapiro, and Patrick Eugster. 2017. Co-design and verification of an available file system. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 358–381.

[11] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: an adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM symposium on Document engineering*. 37–46.

[12] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 259–268.

[13] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. 2009. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 395–403.

[14] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. 2022. DSON: JSON CRDT Using Delta-Mutations for Document Stores. *Proceedings of the VLDB Endowment* 15, 5 (Jan. 2022), 1053–1065. https://doi.org/10.14778/3510397.3510403

[15] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.

[16] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE, 404–412.