DSON: JSON CRDT Using Delta-Mutations For Document Stores

Arik Rinberg*
Technion
Haifa, Israel
ArikRinberg@campus.technion.ac.il

Tomer Solomon
IBM
Tel-Aviv, Israel
tomer.solomon@ibm.com

Roee Shlomo
IBM
Tel-Aviv, Israel
Roee.Shlomo@ibm.com

Guy Khazma
IBM
Tel-Aviv, Israel
Guy.Khazma@ibm.com

Gal Lushi
IBM
Tel-Aviv, Israel
gal.lushi@ibm.com

Idit Keidar
Technion
Haifa, Israel
idish@technion.ac.il

Paula Ta-Shma
IBM
Tel-Aviv, Israel
paula@il.ibm.com

ABSTRACT

We propose DSON, a space efficient δ-based CRDT approach for distributed JSON document stores, enabling high availability at a global scale, while providing **strong** eventual consistency guarantees. We define the semantics of our CRDT based approach formally, and prove its correctness and convergence. Previous approaches optimize for collaborative document editing and store metadata proportional to the number of updates to a document, which is not acceptable for long lived document management. The metadata stored with our approach is bounded by $O(k^2D + n \log n)$, where n is the number of replicas, D is the number of document elements, and $k \leq n$ is the number of concurrent document updates. We also implement our approach[37] and demonstrate its space efficiency empirically. Experimental analysis shows that the metadata stored is typically significantly less than the worst case. This provides the basis for robust highly available distributed document stores with well defined semantics and safety guarantees, relieving application developers from the burden of conflict resolution.

PVLDB Reference Format:
Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. DSON: JSON CRDT Using Delta-Mutations For Document Stores. PVLDB, 15(5): 1053 - 1065, 2022.
doi:10.14778/3510397.3510403

PVLDB Artifact Availability:
The source code and benchmarks have been made available at https://github.com/crdt-ibm-research/json-delta-crdt.

*This work was done during an internship at IBM Research.

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510403

1 INTRODUCTION

NoSQL systems, and in particular document stores, have significantly grown in popularity over the last decade [42], and those in most widespread use adopt a JSON-based data model [41]. Such a data model maps directly to common programming language data structures and relaxes schema restrictions, making it suitable for developers' needs. Applications are diverse but include managing metadata for digital media such as photos and videos, social media posts, product information, and machine-generated data/logs. Distributed document stores are often applied in mobile and edge scenarios where both global scale and high availability under disconnected operation are paramount.

Distributed document stores often default to eventual consistency to enable high availability for both reads and writes on a global scale e.g. CouchDB [5], Couchbase [3], DynamoDB [6]. However, eventual consistency only guarantees that a document's state will converge at some unspecified future point in time if there are no additional updates. Reasoning without substantial safety guarantees adds a burden to application developers and is not suitable for all applications. Conflict resolution policies are often arbitrary [16] and have complex, non-intuitive semantics, e.g., Couchbase offers conflict resolution where the replica with the most mutations wins [4]. To achieve more control over consistency, some document stores provide an API for application developers to manually specify how to merge conflicts when they arise [16]. This approach is time consuming, error prone, and difficult to reason about.

We adopt *Conflict-free Replicated Data Types (CRDTs)* [40] to achieve a distributed document store design with **strong** eventual consistency, meaning that any two nodes that have received the same set of updates will be in the same state. In addition we also provide causal consistency and read-your-writes guarantees. The semantics we propose is well defined and intuitive, enabling programmers to reason about application correctness. We leverage and extend existing theoretical results for CRDTs and apply them to the document store context, where documents can have arbitrarily long lifetimes, but where the amount of metadata used to support CRDTs must be reasonably bounded.

The CRDT approach relies on predetermined conflict resolution rules which dictate their semantics. These rules are typically data structure specific, and CRDTs have been defined for counters, sets, directed graphs etc. [40]. In this paper, we focus on JSON document stores, and treat JSON[9] as a composition of nested maps and arrays (lists) over base data types. Note that since we handle both maps and arrays our work is applicable to other kinds of document stores as well. We define the semantics of our approach formally and prove its convergence and correctness. Existing CRDT approaches for JSON [2, 12] address collaborative document editing and collect metadata proportional to the number of updates. This metadata cannot easily be bounded or cleaned up over time, and therefore these approaches are not suited to our context. Our work enables global high availability for distributed JSON document stores with reasonable overheads and proven semantics.

CRDTs are often split into two main families: op-based [17] and state-based [29]. Op-based approaches are simpler and require smaller messages, but they assume reliable, exactly-once ordered messaging which is hard to maintain even when TCP is used [22]. On the other hand messages in state-based systems include the entire state, which can be prohibitively large. Almeida et al. introduced δ-state-based CRDTs [15] to achieve the best of both worlds – a δ-based CRDT sends a delta instead of the entire state, reducing communication overheads. We chose this approach because it naturally enables bounding metadata growth. However, it requires theoretical guarantees of causal consistency and designing causal δ-based CRDTs is non trivial. The original paper defines several causal δ-CRDTs including multi-value registers, add-wins and remove-wins sets and an enable-wins flag. It also defines a composable (i.e. arbitrarily nestable) Observed-Remove Map which can contain other causal δ-based CRDTs including maps themselves. However, maps on their own are not sufficient to handle JSON[9].

In this paper we define a composable causal δ-based Array CRDT. In addition to arrays (lists) being an important datatype in their own right, this results in a causal δ-based CRDT library powerful enough to capture JSON data. To date various array CRDTs have been proposed, such as Logoot [44], Treedoc [34], WOOT [33] and others [30, 38], as well as a technique to support concurrent move operations on array elements [24]. However none of these achieve composable, causal δ-based array CRDTs. In this paper we define causal composable δ-based array CRDTs supporting arbitrary levels of nesting and concurrent move operations without data duplication.

We base our approach on that of Alameida et. al. [15] and adopt *Observed-Remove (OR)* semantics for Arrays and combine them with their OR-Maps. OR semantics are intuitive because only observed state can be removed. For example, under OR semantics, if there is an update concurrent to a remove, the update wins and the element isn't removed. A shopping cart example is given in Figure 1, where OR semantics is contrasted with Remove-Wins (RW) semantics.

Our main contributions are as follows. We define a δ-based CRDT for Observed-Remove JSON documents which is built on a novel composable δ-based array CRDT. While ORMap semantics was previously defined [40], we formalize the δ-based approach with respect to its execution. We provide a formal definition of the array CRDT, and prove both its convergence and the correctness of its semantics. We show that the amount of metadata stored is bounded by $O(k^2D + n \log n)$, where n is the number of replicas, D is the number of document elements, and is $k \leq n$ is the number of concurrent operations. We implemented a prototype in JavaScript[37] and verify that metadata stays constant with the number of updates, unlike the case for existing JSON CRDTs [2, 12]. Some initial results were presented as an extended abstract[36].

The rest of the paper is structured as follows. Section 2 covers related work, section 3 covers preliminaries, and Section 4 covers our core theoretical results. Section 5 presents the algorithm for the novel Observed-Remove Array CRDT, and Section 6 presents the JSON CRDT. Section 7 surveys our implementation and Section 8 provides an experimental evaluation of metadata growth. Section 9 presents some ideas for future work and Section 10 concludes.

2 RELATED WORK

As described in the introduction, there is no known JSON CRDT that is implemented using delta mutations. The key missing piece is a delta-based array implementation that supports moving of elements and arbitrary nesting, without using tombstones. However, state-based and operation-based constructions do exist. In this section we describe existing CRDTs for arrays and JSON documents as well as describing usage of CRDTs in datastores.

The problem of collaborative text editing has motivated many proposals for array CRDTs such as WOOT [32], Treedoc [35], Logoot [44], and RGA [39]. However, text editing deals with a linear structure and does not consider arbitrary nesting of CRDTs as required for sequences in JSON documents. Higher level constructions, such as the ORArray, are built upon such base constructions.

Moreover, while all mentioned constructions support the insert and delete operations, only RGA supports update and none of them support moving elements. In Moving elements in list CRDTs A generic black box approach to adding move support to existing CRDTs has been proposed [24]. However, the described construction cannot easily be applied in a delta-based setting, e.g., it relies on AWSet that in the delta-based framework is only defined for primitive types and lacks support for an update operation.

A JSON CRDT has been proposed [25] and is implemented in the Automerge library [2]. Automerge is an operation-based CRDT that implements many different data types including composable maps and arrays. It provides a good solution for use cases that require observing all history and supporting undo operations. The array implementation in Automerge is based on RGA, an array CRDT that is implemented similar to a linked list. Each node contains an identifier, an identifier of the next item, a value, and a flag to indicate whether the node is a tombstone. While RGA was originally designed for linear data, nesting is enabled in Automerge by cleverly holding the identifiers of nested CRDTs in the value fields. However, the RGA construction inherently requires tombstones and has an ever-growing metadata problem. Garbage collection cannot be fully implemented without compromising consistency, leading to a significant drawback in use cases such as document stores. In contrast, ORArray and the JSON CRDT built on top of it are naturally implemented without tombstones. Another design choice in Automerge is the handling of conflicts that cannot be resolved automatically. When users concurrently update the same key in the same map (or the same index in the same list) then one of

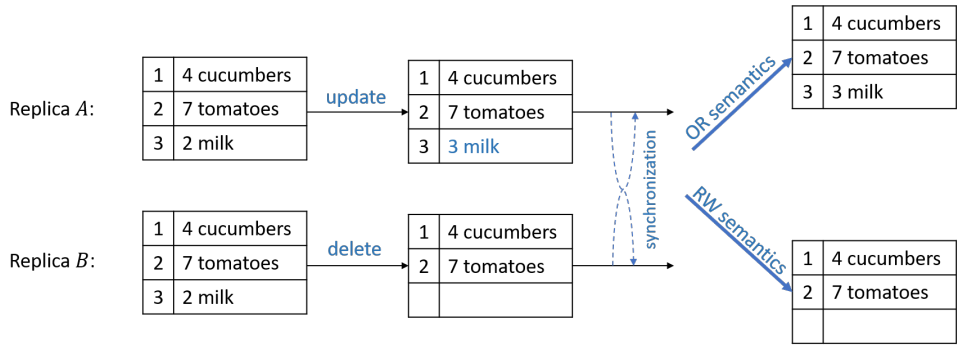


Figure 1: Observed-remove vs. remove-wins semantics via a shopping list example. With OR semantics, only observed state can be removed, therefore the delete at replica B cannot override the update at replica A which had not been observed.

the concurrently written values is picked arbitrarily and the others are stored in a conflicts structure as raw values (i.e., not CRDTs). In contrast, the JSON CRDT presented in this paper preserves all subtrees as valid CRDTs and gives more control to the user, similar to the semantics of a multi-value register, when desired.

Another JSON CRDT has been proposed in YATA [31] and is implemented in the Yjs library [12]. The core of the YATA CRDT is an Array CRDT implemented as a doubly linked list with insert and delete operation. YATA supports update operations and complex data types such as maps by storing *abstract operation types* as the array elements. For example, to support the update operation a *Replace Manager* operation type inserts the updated value to the beginning of an array such that eventually all replicas observe the same first array element as the updated value. While YATA is an operation based CRDT, delete operations are handled using a state based CRDT for storing tombstones. YATA describes a time based garbage collection mechanism and some ad-hoc optimizations are implemented to try minimizing the amount of state stored due to tombstones. The δ-based CRDTs are naturally built without the need for garbage collection, due to the use of a causal context.

δ-based CRDTs have been adopted in industry as part of Akka toolkit [1] for building applications using the actor model. Akka supports delta-based CRDTs for counters, sets, maps and registers but lacks support for arrays. We believe that the addition of a full JSON CRDT can both open the door for new use cases and greatly simplify application development.

δ-based CRDTs are also used to build real-time collaborative applications on top of IPFS [10, 18]. The array implementation is based on RGA with tombstones and without any support for nesting. That is, you can't store an array inside a map nor store any CRDT type inside the array.

Distributed databases such as Redis, Riak, Cosmos DB and Antidote DB include support for CRDT data types such as sets and maps. However, for data stores such as CouchDB that use JSON as the data model a full JSON CRDT is required. AutoCouch [21] leverages both the AutoMerge library and CouchDB to enable developing collaborative applications. AutoCouch synchronizes the entries of the AutoMerge history using the database's revision tree. As stated by the authors the main issue is the ever growing metadata size, both due to tombstones and due to the inherent history keeping in Automerge. We believe that our approach provides an alternative that preserves all of the functionality required by document stores while keeping the synchronization cost low due to the use of delta mutations and a design that does not require garbage collection. Our approach may also be suitable for other use cases that use the JSON model, such as decentralized Kubernetes control planes [23, 28].

3 PRELIMINARIES

Section 3.1 presents our model, and in Section 3.2 we define our notations. In Section 3.3 we present background δ-based CRDTs.

3.1 Model

Our system consists of a set of *nodes* $\mathcal{I}$ hosting *replicas* of objects; we use node and replica interchangeably. The object exposes API *methods* which are used to alter the underlying state of the object. Nodes communicate with each other by *broadcasting* messages over unreliable, unordered links. The network provides the following two assumptions: (1) If a node delivers message m, then another node previously broadcast message m, and (2) if a message is broadcast infinitely many times it is eventually delivered by every node. We denote a broadcast of message m by node i as $bcast_i(m)$, and deliver of message m broadcast by node i at node j as $deliver_j(m, i)$. An *event* occurring at a node is either an API method invocation, a message broadcast, or a deliver.

An *execution* σ of the system is an interleaving sequence of events and states. A *trace* of an execution σ is the sub-sequence of events in σ. We denote the trace of an execution σ as $Tr(\sigma)$.

We make use of the *happens-before* relation as defined by Lamport [27]. An event $e_1 \in Tr(\sigma)$ happens-before or *precedes* an event $e_2 \in Tr(\sigma)$ in an execution σ, denoted $e_1 \prec_\sigma e_2$ if: (1) e_1 occurs before e_2 in the same replica in $Tr(\sigma)$, or (2) e_1 is a broadcast of message m and e_2 is a deliver of that message, or (3) there exists some event $e \in Tr(\sigma)$ such that $e_1 \prec_\sigma e$ and $e \prec_\sigma e_2$. We define the set of operations in σ as $Ops(\sigma) = \{e \in Tr(\sigma) | e \text{ is an API method invocation}\}$. We say that two events e_1 and e_2 are *concurrent* if neither precedes the other in σ, denoted $e_1 || _\sigma e_2$.

Previous work defines CRDT object semantics in terms of the poset $(Ops(\sigma), \prec_\sigma)$ [29], i.e., the set of operations and the corresponding happens-before partial order. As an example, we present

the *Multi-Value Register (MVReg)*, a CRDT register. The register exposes the following API methods: (1) WRITE(x), and (2) READ(). Given an execution σ, the READ method returns:

$$\{v | \exists \text{WRITE}(v) \in Ops(\sigma) \wedge \nexists \text{WRITE}(u) : \text{WRITE}(v) \prec_\sigma \text{WRITE}(u)\}.$$

For example, if node i executes WRITE(v_1) and concurrently node j executes WRITE(v_2), a READ operation on node k that is preceded by both these WRITE operations returns $\{v_1, v_2\}$. If node k executes a WRITE(v_3) after the READ, then an ensuing READ returns $\{v_3\}$.

3.2 Notations

We consider standard notations for sets and maps. The sets we consider in this paper are partially ordered (namely, posets), with a least element $\bot$. A *map* is a set of key-value pairs $\{k \mapsto v\}$, where each key k is from some set $\mathcal{K}$, and is associated with a single value v from some set $\mathcal{V}$. Given a map m and a key $k \in \mathcal{K}$, we denote its associated value by $m[k]$. The domain of a map m is denoted $\texttt{dom}\ m$ and the range of a map m as $\texttt{ran}\ m$, i.e., $\texttt{dom}\ m = \{k \mid \{k \mapsto v\} \in m\}$ and $\texttt{ran}\ m = \{v \mid \{k \mapsto v\} \in m\}$. For a map m and a key k, if $k \notin \texttt{dom}\ m$, we say that $m[k] = \bot$, i.e., all keys not present in the map are mapped to $\bot$. For a pair $p = (a, b)$, we use $fst(p)$ and $scnd(p)$ to denote its first (a) and second (b), respectively. Finally, for a set S we denote its power set by $P(S)$.

A *join semilattice* [19], or simply a *lattice*, is a three tuple $(S, <, \sqcup)$, where S is a set, $<$ is a partial order on S, and $\sqcup$ is a function $S \times S \mapsto S$ that is associative, commutative, and idempotent.

Table 1: Table of notations.

Notation	Explanation
$\mathcal{I}$	Set of nodes
$bcast_i(m)$	Broadcast event of message m by node $i \in \mathcal{I}$
$deliver_j(m, i)$	Deliver event of message m broadcast by node $i \in \mathcal{I}$, at node $j \in \mathcal{I}$
σ	An execution of the system, including events and states.
$Tr(\sigma)$	Sub-sequence of events (trace) of execution σ.
$e_1 \prec_\sigma e_2$	Event e_1 precedes event e_2 in execution σ
$e_1 \| \|_\sigma e_2$	Event e_1 and e_2 are concurrent in execution σ
$Ops(\sigma)$	The set of API methods invoked in σ.

3.3 δ-Based CRDTs

As mentioned in the introduction, δ-based CRDTs were proposed by Almeida et al. [15] to achieve the best of both state-based CRDTs and op-based CRDTs. This is achieved by using a join semilattice. The state of the CRDT and the broadcasted messages are both elements of the join semilattice. We denote the broadcasted messages as *deltas*.

The API methods use the current object state X to generate a delta X^δ that represents the incremental movement from the old state (before the method invocation) to the new one (after), i.e., X is the old state and $X \sqcup X^\delta$ is the new one. After generating X^δ, the node joins its state with X^δ to get the new state, and broadcasts X^δ to all nodes.

Nodes needn't keep track of which messages have been received, as the join is idempotent, and don't need to deliver messages in the same order, as the join is commutative and associative. Finally, convergence is achieved whenever all nodes deliver all messages from all other nodes. An *anti-entropy algorithm* [14] is used to enforce convergence and preserve causality: it retransmits outstanding deltas, enforces the causal order of delivery, and merges outstanding deltas in order to send fewer messages.

Commonly, δ-based CRDTs assign unique identifiers to operations. The simplest way to generate these identifiers is for replica $i \in \mathcal{I}$ to generate the sequence of pairs $(i, 1), (i, 2), \ldots$ and assign a pair per event. We denote such a pair a *dot*. An operation's dot is broadcast along with the delta reflecting its mutation. We say that a dot has been *observed* by replica i if: (1) i generated the dot, or (2) i delivered a message containing the dot. The collection of observed events at a node up to a given point in an execution is called a *causal context*, and it is maintained locally at every replica as part of its state. The causal context c local to i exposes three functions: $\texttt{max}_i(c)$, which gives the maximum sequence number in the causal context, $\texttt{next}_i(c)$, which produces the next dot for replica i, and $\texttt{insert}_i(c, d)$, which adds dot d to the causal context. The causal context is defined in Equation 1.

$$\begin{gathered} \textsc{CausalContext} \in P(\mathcal{I} \times \mathbb{N}) \\ \texttt{max}_i(c) = \max(\{n \mid (i, n) \in c\} \cup \{0\}) \\ \texttt{next}_i(c) = (i, \texttt{max}_i(c) + 1) \\ \texttt{insert}_i(c, d) = c \cup \{d\} \end{gathered} \tag{1}$$

As the causal context is a grow-only set, its size is unbounded. In practice, it can be efficiently compressed without loss of information. As dots are created sequentially, instead of maintaining a list of dots $(i, 1), (i, 2), \ldots, (i, n)$ for every replica i, we can maintain a mapping $prefix : \mathcal{I} \mapsto \mathbb{N}$, such that if $prefix[i] = n$ then all dots (i, m) for $m \leq n$ are in the causal context. Due to the unreliable network, we may have a gap in the sequence, e.g., replica j may observe $(i, 1)$ and $(i, 3)$ but not $(i, 2)$. We therefore also maintain a separate (uncompressed) set of dots exceeding the longest prefix.

A DOTSTORE is a container for data-type specific information and is used to store the state of a δ-based CRDT. It provides the operation `dots` for querying the set of event identifiers (i.e., dots) currently stored in the dot store. We now present the dot stores defined by Almeida et al. [15]. The DOTFUN instantiated with lattice V is a map from dots in $\mathcal{I} \times \mathbb{N}$ to V. Its `dots` operation returns the domain of the map. It is presented formally in Equation 2.

$$\begin{gathered} \textsc{DotFun}\langle V : Lattice \rangle : \mathcal{I} \times \mathbb{N} \mapsto V \\ \texttt{dots}(s) = \texttt{dom}\ s \end{gathered} \tag{2}$$

The DOTMAP instantiated with a set K and a DOTSTORE V is a map from K to V. Its `dots` operation returns the union of the return values of invoking the `dots` operation on each DOTSTORE in the range. Equation 3 presents it formally.

$$\begin{gathered} \textsc{DotMap}\langle K, V : DotStore \rangle : K \mapsto V \\ \texttt{dots}(m) = \bigcup_{k \in \texttt{dom}\ m} \texttt{dots}(m[k]) \end{gathered} \tag{3}$$

To form the join semilattice that makes up the state and deltas of δ-based CRDTs, dot stores are paired with causal contexts. We denote the pair DotStore×CausalContext as Causal⟨DotStore⟩. For each type of Causal⟨DotStore⟩ we define the join function ($\sqcup$). The semilattice's partial ordering is defined with respect to the join function: For any $X \in$ Causal⟨DotStore⟩, $X \sqcup \bot = X$. For two elements $X_1, X_2 \in$ Causal⟨DotStore⟩ we say that $X_1 < X_2$ iff $\exists X \neq \bot \in$ Causal⟨DotStore⟩ such that $X_1 \sqcup X = X_2$.

The merge in the Causal⟨DotFun⟩ keeps values that exist in both of the mappings and merges their respective values, or exist in either one of the mappings and are "new" to the other, in the sense that they are not in its causal history. It is formally defined in Equation 4.

$$\begin{aligned}(m,c) \sqcup (m',c') = \\ (\{d \mapsto m[d] \sqcup m'[d] \mid d \in \mathsf{dom}\ m \cap \mathsf{dom}\ m'\} \\ \cup \{(d,v) \in m | d \notin c'\} \cup \{(d,v) \in m' \mid d \notin c\}, c \cup c')\end{aligned} \tag{4}$$

The merge in the Causal⟨DotMap⟩ applies the merge recursively on each of the keys in either domains, and keeps all none $\bot$ values. It is presented formally in Equation 5. An example of a $\bot$ value is a merge between two elements of the Causal⟨DotFun⟩ semilattice, where the domains are disjoint but all mappings are in the others causal history. Consider for example a write w_1 that precedes a write w_2, i.e., $w_1 \prec_\sigma w_2$, then the dot generated by w_1 is in the causal context of the delta generated by w_2. By the definition of join, the mapping doesn't "survive" the join, and therefore the old value (written by w_1) is overwritten – it isn't present in the range of the map after w_2.

$$\begin{aligned}(m,c) \sqcup (m',c') = (\{k \mapsto v(k) \mid k \in \mathsf{dom}\ m \cup \mathsf{dom}\ m' \\ \wedge v(k) \neq \bot\}, c \cup c') \\ \textbf{where } v(k) = fst((m(k),c) \sqcup (m'(k),c'))\end{aligned} \tag{5}$$

We next give an example of a δ-based CRDT, via the MVReg [15]. Its pseudo-code is presented in Algorithm 1. The state is an element (m,c) in the semilattice Causal⟨DotFun⟩. Namely, a mapping from dots to values in V. The write(v) operation creates a delta $X^\delta = (m^\delta, c^\delta) \in$ Causal⟨DotFun⟩ as follows: c^δ contains the current domain of the DotFun, which is the set of dots written but not over-written, and m^δ is a mapping from a new dot to the written value v. Note that any preceding write(u) that is contained m before the operation has its dot added to c^δ, therefore, when joining X^δ with the current state, u is removed from m. A read operation returns all values that are not removed. For example, consider two replicas i and j of a new MVReg. Replica i executes write$_i(v)$, generates delta $(\{(i,1) \mapsto v\}, \{(i,1)\})$, merges and broadcasts it. Replica j delivers the delta, and changes its state to $(\{(i,1) \mapsto v\}, \{(i,1)\})$. Now, replica i executes write$_i(u)$ concurrently to replica j executing write$_j(w)$. The delta generated by write$_i(u)$ is $(\{(i,2) \mapsto u\}, \{(i,1),(i,2)\})$, and the delta generated by write$_j(w)$ is $(\{(j,1) \mapsto w\}, \{(i,1),(j,1)\})$. After all deltas have been merged, both replicas are in the state:

$$(\{(i,2) \mapsto u, (j,1) \mapsto w\}, \{(i,1),(i,2),(j,1)\}),$$

so the MVReg holds both values u and w.

Algorithm 1 MVReg algorithm for node i.

```
1: Global state (m, c)                                  ▷ Causal⟨DotFun⟩
2: procedure write_i(v)
3:     d ← next_i(c)                                    ▷ New dot
4:     c' ← dom m                                       ▷ Currently stored dots
5:     return ({d ↦ v}, c' ∪ {d})                       ▷ New X^δ
6: procedure read_i
7:     return ran m
```

4 OBSERVED-REMOVE SEMANTICS

Observed-remove semantics are a commonly used in production systems (e.g., Riak [26]). Under such semantics, an operation can overwrite ("undo") preceding operations executed on the same element. The MVReg is a simple example of such an object, where a write overwrites any preceding write operations albeit not concurrent ones: If two write operations are concurrent, then both values are reflected in the state. We now present two more elaborate examples, and formalize their OR semantics.

4.1 δ-Based ORMap

We first formalize the *Observed Remove Map (ORMap)* [15]. It is a mapping of some set of keys K, each to an arbitrary δ-based CRDT, possibly an ORMap itself. Thus, the ORMAP is *composable*. The map has no schema, in that each key can be mapped to a different type of CRDT. Its API consists of the following methods:

- apply(k, o_i^δ) – given a key $k \in K$ mapped to some δ-based CRDT v of type V, and a method o_i^δ from the API of V, The method applies o_i^δ to the CRDT by invoking o_i^δ on v.
- delete(k) – deletes key k from the map.
- get(k) – returns the embedded value of the CRDT mapped to by k.

For example, consider a map from keys to MVRegs. Writing value v to key k_1 is done by executing apply$(k_1,$ write$(v))$ to the map, and get(k_1) reads the same MVReg.

Recall that the deltas generated by API methods and the replica states are both members of the same join semilattice. We next define the semantics of the δ-based ORMap. Given execution σ, define:

$$\begin{aligned}\mathcal{X}_{apply}(\sigma) \triangleq \{X^\delta | \exists \text{apply}_i(k, o_i^\delta) \in Ops(\sigma) \text{ that returns } X^\delta \\ \wedge \nexists \text{delete}_j(k) \in Ops(\sigma) : \text{apply}_i(k, o_i^\delta) \prec_\sigma \text{delete}_j(k)\}\end{aligned}$$

$$X(\sigma) \triangleq \begin{cases} \bigsqcup_{X^\delta \in \mathcal{X}_{apply}(\sigma)} X^\delta & \mathcal{X}_{apply}(\sigma) \neq \emptyset \\ \bot & otherwise \end{cases}$$

Intuitively, $\mathcal{X}_{apply}$ is the set of deltas generated by apply operations whose keys were not subsequently deleted, and X is the join of these deltas.

Furthermore, assume the existence of an abstract function Value(X,k), which retrieves the value of key k from any element X in the join semilattice, similarly to executing get(k) on a replica with state X. Then a get(k) returns Value$(X(\sigma),k)$.

Under this definition, in the presence of a concurrent apply and delete of the same key, the delete removes all apply operations that causally precede it and leaves the key in the map with the

concurrently written value. A DELETE can thus be seen as an "undo" of all operations leading up to it.

The semantics of the ORMap were previously defined less formally in [15]. Here we formally define the semantics of the return value of a GET for the first time.

Algorithm 2 presents the ORMap pseudo-code, as proposed in [15]. The state is an element in Causal⟨DotMap⟩. The causal context is shared by all keys and corresponding nested CRDTs. The APPLY method fetches the value at key k from m and pairs it with the shared causal context c to obtain the value from the embedded type. It then invokes the operation over the pair. From the resulting pair it extracts the value to create a new mapping for that key, which it pairs with the resulting causal context. As the DELETE method uses the `dots` method from within a DotMap, the dots returned are the dots of all embedded values, therefore deleting a key recursively removes the corresponding embedded values.

Algorithm 2 ORMap algorithm for node i.

```
1: Global state (m, c)                          ▷ Causal⟨DotMap⟩
2: procedure APPLY_i(k, o_i^δ)
3:     (v, c') ← o_i^δ(m[k], c)                 ▷ Apply the mutator.
4:     return ({k ↦ v}, c')
5: procedure DELETE_i(k)
6:     c' ← dots(m[k])                          ▷ Get all embedded dots.
7:     return (⊥, c')
```

4.2 δ-Based ORArray

We now introduce a new OR data type, the *Observed Remove Array (ORArray)*. To this end, we represent an array as a set of value-position pairs, whose values are general CRDTs. Using an integer index as the position would require updating multiple element positions following an insertion or deletion. Therefore, we adopt the notion of *stable identifiers* [24], which allow insertion and deletion without updating other elements, for example, the set of real numbers. An insertion of an element between an element at position p_1 and one at p_2, inserts an element at position $(p_1 + p_2)/2$. The array is the sequence of values sorted by the corresponding position in ascending order. For example, the array $[a, b, c]$ can be represented by the set $\{(a, 1.0), (b, 7.5), (c, 42.7)\}$.

As the ORArray allows concurrent operations, the position may be ambiguous, similarly to the value of an MVReg under concurrent writes. To this end, we extend the position to be a set of possible positions. For example, if element b is concurrently moved before a and after c, it's position may be $\{0.3, 54\}$. To sort the array, replicas deterministically choose an element from the position set (e.g., the maximum) to use for ordering the values. To keep track of values as they change positions under concurrent operations, we identify each array element by a unique identifier. For example, the unique identifier can be the dot with which the element was created (as the creation point is unique). The representation of the ORArray is therefore a map from unique identifiers to value-positions pairs. For example, the aforementioned array can be represented as the map: $\{(i, 1) \mapsto (a, \{1.0\}), (j, 1) \mapsto (b, \{0.3, 54\}), (i, 2) \mapsto (c, \{42.7\})\}$, where $(i, 1)$ is the dot marking the creation even of element a, $(j, 1)$ is the creation event of b, and $(i, 2)$ is the creation event of c. When an element is inserted for the first time, the API receives the unique identifier that identifies the inserted element until its deletion.

The value of an element can be any δ-based CRDT, possibly an ORArray itself. Thus, the ORArray is also composable. The array has no schema; each value can be a different type of CRDT.

Its API consists of the following methods:

- **APPLY**(uid, o_i^δ, p) – given a unique identifier uid mapped to some pair consisting of a δ-based CRDT v of type V and a set of positions, a method o_i^δ from the API of V, and a target position p, The method applies o_i^δ to the CRDT by invoking o_i^δ on v, and overwriting the set of positions with p.
- **MOVE**(uid, p) – given a unique identifier uid and a position p, overwrites the position of the element to be p.
- **DELETE**(uid) – deletes any element mapped to uid from the map.
- **GET**(uid) – given a uid mapped to a value v of some CRDT of type V and a set of positions P, returns the value of v based on V paired with the set of positions P.

While the API uses a unique id to refer to an element in an array and uses stable position identifiers to sort it, traditional use of an array consists of using integer indices, i.e., the first element of an array arr is $arr[0]$. To this end, we also provide a wrapper API and implemented functions converting an integer index to a unique id and to a stable position identifier. Converting an index idx to a unique identifier uid is done by sorting the array using the position identifiers, and returning the uid of the element at index idx. Converting an index to a position identifier is done in a similar fashion. The wrapper API consists of the following methods, mapped to the methods above:

- **INSERT**(idx, o_i^δ) – given an index idx and a method o_i^δ from the API of some CRDT of type V, The method assigns a unique id uid, assigns a stable position identifier p such that the new element in the sorted array appears at index idx, and invokes APPLY(uid, o_i^δ, p).
- **UPDATE**(idx, o_i^δ) – given an index idx and a method o_i^δ of some CRDT type V, The method finds the uid corresponding to the element at index idx, finds the position p, and invokes APPLY(uid, o_i^δ, p).
- **MOVE**(old_idx, new_idx) – given two indexes, finds the element uid corresponding to the element at index old_idx, calculates the stable position identifier p such that the element in the sorted array will be at index new_idx, and invokes MOVE(uid, p).
- **DELETE**(idx) – given an index idx, finds the element uid corresponding to index idx, and invokes DELETE(uid).
- **GET**(idx) – given an index idx, finds the element uid corresponding to the element at index idx, and invokes GET(uid).

We define the semantics of the ORArray in two steps: First the value and next the position. We define the semantics of the value

in a similar fashion to Section 4.1. Given execution an σ, define:

$$\mathcal{X}_{apply}(\sigma) \triangleq \{X^{\delta} | \exists \textsc{Apply}_i(uid, o_i^{\delta}, p) \in Ops(\sigma) \text{ that returns } X^{\delta}$$
$$\wedge \nexists \textsc{Delete}_j(uid) \in Ops(\sigma) : \textsc{Apply}_i(uid, o_i^{\delta}, p) \prec_{\sigma} \textsc{Delete}_j(uid)\}$$

$$X(\sigma) \triangleq \begin{cases} \bigsqcup_{X^{\delta} \in \mathcal{X}_{apply}(\sigma)} X^{\delta} & \mathcal{X}_{apply}(\sigma) \neq \emptyset \\ \bot & otherwise \end{cases}$$

Furthermore, assume the existence of an abstract function which gets the value of element *uid* from any element X in the join semilattice, similarly to executing $\textsc{get}(uid)$ on a replica with state X, denoted $\textsc{Value}(X, uid)$. Then $fst(\textsc{get}(uid))$ returns $\textsc{Value}(X, uid)$.

To define the positional semantics, we first identify the subset of $Ops(\sigma)$ consisting of all operations that alter element *uid* (i.e., $\textsc{apply}(uid, o_i^{\delta}, p)$, $\textsc{move}(uid, p)$, and $\textsc{delete}(uid)$), and do not precede another operation that alters element *uid* in σ. We denote this subset as $maximal(\sigma, uid)$.

Next, given an execution σ, define:

$$P_{apply}(uid, \sigma) \triangleq \{p | \exists \textsc{Apply}(uid, o_i\delta, p) \in maximal(\sigma, uid)\}$$
$$P_{move}(uid, \sigma) \triangleq \{p | \exists \textsc{Move}(uid, p) \in maximal(\sigma, uid)\}$$

At the end of σ, if $fst(\textsc{get}(uid)) \neq \bot$ then $scnd(\textsc{get}(uid))$ returns a non-empty set of positions $P(uid)$ such that:

$$P_{apply}(uid, \sigma) \subseteq P(uid) \subseteq \left(P_{apply}(uid, \sigma) \cup P_{move}(uid, \sigma)\right).$$

Note that if $P_{apply}(uid, \sigma)$ is empty, and the value is not $\bot$, then the position has at least one element from $P_{move}(\sigma)$.

Under these semantics, the position set of an element contains all positions set by APPLY operations, as such operations update the value and therefore it is important they not be overwritten. It may or may not contain positions set by MOVE operations that occur concurrently with other updates, as these do not impact the underlying data.

5 ORARRAY ALGORITHM

In Section 5.1 we present a new dot store, and in Section 5.2 we present our algorithm implementing the δ-based ORArray.

5.1 The CompDotFun

To construct the ORArray we first propose a new dot store. We combine the DotMap and DotFun to get a dot store which maps dots to dot stores. We assume the dot stores in the range of the DotFun are all join-semilattices. We will show that the resulting mapping is also a join-semilattice. Thus, this construction yields a composable DotFun. We denote this dot store as CompDotFun. The dot store is presented in Equation 6. Note that the `dots` method returns the dots in the domain, as well as a union on recursive calls of `dots` on all dot stores in the range.

$$\textsc{CompDotFun}\langle V : DotStore\rangle : \mathcal{I} \times \mathbb{N} \mapsto V$$
$$\texttt{dots}(m) = \texttt{dom}\ m \cup \bigcup_{v \in \texttt{ran}\ m} \texttt{dots}(v) \tag{6}$$

We define its corresponding join-semilattice in Equation 7. The join operation keeps keys that have not been deleted (as in the DotFun), or the values themselves haven't been deleted (as in the DotMap).

$$\begin{gathered}(m, c) \sqcup (m', c') = \\ (\{d \mapsto v(d) \mid d \in \texttt{dom}\ m \cap \texttt{dom}\ m' \wedge v(d) \neq \bot\} \\ \cup \{(d, v) \in m \mid d \notin c'\} \cup \{(d, v) \in m' \mid d \notin c\}, c \cup c') \\ \textbf{where } v(d) = fst((m(d), c) \sqcup (m'(d), c'))\end{gathered} \tag{7}$$

It can be proven that given three elements in $Causal\langle\textsc{CompDotFun}\rangle$, the resulting object is depends on set union and the join of the inner dot store of the three elements. As set union is commutative, associative, and idempotent, and as we assume that the inner dot store is a join semilattice we arrive at the following corollary:

COROLLARY 5.1. *The $Causal\langle\textsc{CompDotFun}\rangle$ is a join-semilattice.*

5.2 The ORArray

We now propose an algorithm implementing the δ-based ORArray. The ORArray is instantiated with some set of stable position identifiers $\mathcal{P}$. Algorithm 3 presents our proposal. We store the position in the following dot store:

$$\textsc{Pos} \triangleq \textsc{CompDotFun}\langle\textsc{DotFun}\langle\mathcal{P}\rangle\rangle$$

The state and deltas are generated by the algorithm are from the following join-semilattice:

$$\textsc{Causal}\langle\textsc{DotMap}\langle\mathcal{I} \times \mathbb{N}, (DotStore, \textsc{Pos})\rangle\rangle.$$

In other words, the ORArray is stored as a mapping from dots to pairs of value-positions, where the value is any CRDT, and the set of positions is stored using a DotFun nested in a CompDotFun. This position construction is what allows for concurrent updates. This construction stores the set of positions as a forest, where each tree is of depth 3. For example, an element whose set of positions is $\{0.3, 54\}$ may have the position stored in the following forest:

$$\{d_1 \mapsto \{d_2 \mapsto 0.3\}, d_3 \mapsto \{d_4 \mapsto 54\}\},$$

where d_1, d_2, d_3, and d_4 are dots. The APPLY function adds a new tree and removes all the old ones by removing their roots. In the example above, the roots d_1 and d_3 are removed and a new tree is inserted. As the root is stored in a CompDotFun, once a root is removed it never reappears, contrary to keys in a DotMap which may remain undeleted if there is a concurrent update. The MOVE function removes the child of all roots and adds a single child to every root with the new position. Consider the example above where the position is set to 42. The dots d_2 and d_4, and the forest is altered to

$$\{d_1 \mapsto \{d_5 \mapsto 42\}, d_3 \mapsto \{d_5 \mapsto 42\}\}.$$

The DELETE function removes all roots, thus removing the position. A MOVE may be overwritten by a concurrent APPLY or DELETE, as the APPLY and DELETE remove the roots. But an APPLY always adds a new root, thus an element cannot have a non-empty value and be without any possible position.

As an APPLY inserts a new tree into the forest, concurrent APPLY operations create multiple trees. For example, if one apply sets the position to $\{d_1 \mapsto \{d_2 \mapsto 0.3\}\}$ and another one sets the position to $\{d_3 \mapsto \{d_4 \mapsto 54\}\}$, after they are merged the resulting forest is the one appearing above.

We now prove that Algorithm 3 implements an ORArray. We begin with the values. Note that the implementations of the $\textsc{apply}_i$

Algorithm 3 ORArray algorithm for node i.

```
1:  Global state (m, c)                          ▷ Causal⟨DotMap⟨I ×
    ℕ, (DotStore, CompDotFun⟨DotFun⟨P⟩⟩)⟩⟩
2:  procedure apply_i(uid, o_i^δ, p)
3:      d ← next_i(c)
4:      (v, c') ← o^δ(fst(m[uid]), c ∪ {d})
5:      roots ← {d' | d' ∈ dom snd(m[d])}        ▷ Get all roots
6:      m' ← {uid ↦ (v, {d ↦ {d ↦ p}})}
7:      c' ← c' ∪ {d} ∪ roots
8:      return (m', c')
9:  procedure move_i(uid, p)
10:     d ← next_i(c)
11:     roots ← {d' | d' ∈ dom snd(m[d])}        ▷ Get all roots
12:     children ← ⋃_{child ∈ dom (ran snd(m[uid]))} dots(child)
13:     position ← {r ↦ {d ↦ p} | r ∈ roots}
14:     m' ← {uid ↦ ({}, position)}
15:     c' ← {d} ∪ children
16:     return (m', c')
17: procedure delete_i(uid)
18:     c' ← dots(m[uid])
19:     return ({}, c')
```

and the DELETE_i operations in Algorithm 3 are, disregarding Line 3 and Line 5, identical to those in Algorithm 2. The difference in the APPLY_i operation is due to updating the position identifier. Furthermore, the semantics as defined in Section 4 are identical. Therefore, we begin with the following observation:

Observation 1. *The value of an element in an object implemented by Algorithm 3 adheres to OR semantics.*

We next show that the set of positions of an element P satisfies the semantics described in Section 4.2. We show each containment separately.

Lemma 5.2. *Consider an execution σ of Algorithm 3. Let P be the set of possible positions of some element at the end of σ. Then $P_{apply}(uid, \sigma) \subseteq P$.*

Proof. If $p \in P_{apply}(uid, \sigma)$, then there exists some $\text{APPLY}(uid, o_i^\delta, p)$ in $maximal(\sigma, uid)$. When $\text{APPLY}(uid, o_i^\delta, p)$ executes, it execute Line 3, generating a new dot d. It then adds $\{d \mapsto \{d \mapsto p\}\}$ as the position in the delta it generates X^δ. As the operation is a maximal operation, no other operation has seen dot d. In particular, the local causal context does not contain d. Therefore, the position survives the join, and $\{p \mapsto \{p \mapsto d\}\} \in scnd(\text{GET}(uid))$ for a GET that was to execute at this point. Therefore, $p \in P$.

This is true for all $p \in P_{apply}(uid, \sigma)$, and therefore:

$$P_{apply}(uid, \sigma) \subseteq P.$$

□

We now show the right hand side.

Lemma 5.3. *Consider an execution σ of Algorithm 3. Let P be the set of possible positions of some element at the end of σ. Then $P \subseteq P_{apply}(uid, \sigma) \cup P_{move}(uid, \sigma)$.*

Proof. We prove the lemma by showing that if $p \notin P_{apply}(uid, \sigma) \cup P_{move}(uid, \sigma)$, then $p \notin P$ for any p. Consider some $p \notin P_{apply}(uid, \sigma) \cup P_{move}(uid, \sigma)$. Trivially, if there exists no operation in $Ops(\sigma)$ that sets the position to p, then $p \notin P$.

Consider the case that there exists either a $\text{APPLY}(uid, o_i^\delta, p) \in Ops(\sigma)$ or $\text{MOVE}(uid, o_i^\delta, p) \in Ops(\sigma)$. Denote the operation as o_p. As $p \notin P_{apply}(uid, \sigma) \cup P_{move}(uid, \sigma)$, then $o_p \notin maximal(\sigma, uid)$. Therefore, there exists some other operation $o \neq \text{GET} \in Ops(\sigma)$ such that $o_p \prec_\sigma o$.

When o_p returns a delta X^δ, then the position in X^δ is of the form $\{d_1 \mapsto \{d^1 \mapsto p\}, d_2 \mapsto \{d^2 \mapsto p\}, \ldots d_k \mapsto \{d^k \mapsto p\}\}$ for some integer k. We now show that no matter what type of operation o is, p is removed after o. If o is an APPLY, then on Line 5 dots $\{d_1, d_2, \ldots, d_k\}$ are added to the causal context and not added to the map. Denote the delta returned by o as $X^{\delta\prime}$. Then when X^δ is joined with $X^{\delta\prime}$, these dots do not survive that join, as they are keys in a CompDotFun. If o is a DELETE, then dots $\{d_1, d_2, \ldots, d_k\}$ are added to the causal context and the result is identical. If o is a MOVE, then on Line 12 dots $\{d^1, d^2, \ldots, d^k\}$ are added to the delta's causal context. As this join is in a DotFun, the dots don't survive the join.

In either one of the cases, the position does not survive the execution of o, and therefore $p \notin P$. □

Lemma 5.4. *Consider an execution σ of Algorithm 3. Let P be the set of possible positions of some element at the end of σ, and denote the value of the element v. If $v \neq \bot$ then $P \neq \emptyset$*

Proof. Consider some execution σ of Algorithm 3, and consider some element uid, such that its corresponding value v is not $\bot$. If $P_{apply}(uid, \sigma) \neq \emptyset$ then the lemma is proven, as $P_{apply}(uid, \sigma) \subseteq P$. Assume, therefore, that $P_{apply}(uid, \sigma) = \emptyset$.

As $v \neq \bot$, there exists some APPLY operation that does not precede a DELETE in σ. Consider the maximal such APPLY operation, i.e., an APPLY that does not precede a DELETE or another APPLY operation in σ. Let d be the new dot generated on Line 3 during the execution of the operation. As $P_{apply}(uid, \sigma) = \emptyset$, then there exists a MOVE operation such that the APPLY precedes the MOVE in σ. Consider the maximal such MOVE in σ. During the execution of the operation, the MOVE adds $\{d \mapsto \{d' \mapsto p\}\}$ to the position map. As d wasn't yet deleted (as it is only removed due a subsequent APPLY or DELETE), and d' is a new dot generated by the MOVE, the local state satisfies: $\{d \mapsto \{d' \mapsto p\}\} \in scnd(m[uid])$. Therefore $p \in P$, and $P \neq \emptyset$. □

6 DSON

We now describe the construction of DSON. A JSON document [9] is defined recursively as follows:

$$\begin{aligned} \text{Register} &= String|Number|null \\ \text{Map} &= \{String : \text{Value}, \ldots, String : \text{Value}\} \\ \text{Array} &= [\text{Value}, \text{Value}, \ldots, \text{Value}] \\ \text{Value} &= \text{Map}|\text{Array}|\text{Register} \end{aligned}$$

A JSON document is a Value itself. The identifier of a Value in a JSON document is the path followed from the root to get to the

VALUE, starting from *doc*. For example, given the JSON document:

$$\{k_1 : v_1, k_2 : [v_2, v_3], k_3 : \{k_4 : v_4\}\},$$

the identifier of v_1 is $doc.k_1$, the identifier of v_3 is $doc.k_2[1]$, and the identifier of the value v_4 is $doc.k_3.k_4$.

The JSON document is one of three objects: REGISTER, MAP, and ARRAY. We define each object's API methods:

- **REGISTER:**
  (1) WRITE(x) – sets register value to x;
  (2) READ() – returns the register value.
- **MAP:**
  (1) APPLY(k,o^i_δ) – given a key k and function o^δ_i, applies o^δ_i to key k;
  (2) GET(k) – returns the VALUE of key k;
  (3) REMOVE(k) – removes key k from the map.
- **ARRAY:**
  (1) APPLY(uid,o^i_δ,p) – given a unique identifier uid, function o^δ_i and position p, applies o^δ_i to element uid and sets its position to p;
  (2) MOVE(uid, o^i_δ, p) – given a unique identifier uid and position p, set the position of element uid to p;
  (3) GET(uid) – returns the VALUE of element uid;
  (4) REMOVE(uid) – removes the element uid.

An *Observed-Remove JSON (OR-JSON)* CRDT is one in which the register, map, and array all adhere to OR semantics, as defined in Section 4. We achieve this by implementing the register using the MVReg, the map using the ORMap, and the array using the ORArray. The replicas communicate with each other via a causal anti-entropy algorithm, and each replica is sequential.

Our solution provides the following guarantees:

- *Read-your-writes* [43] – if a read method is executed following a write method in the same replica, then the read sees the write;
- *Causal Consistency* [27] – if a read method sees a write w_1, and w_2 causally precedes w_1, then the read sees w_2;
- *Strong Eventual Consistency* – all nodes eventually converge on the same state, such that identical reads on different nodes return identical values.

Read-your-writes is achieved because every replica is sequential, an it merges the new delta with its local state one an operation returns, before any subsequent reads. Eventual consistency is achieved by (1) using some anti-entropy algorithm that ensures that the latest states or updates eventually reach all replicas, e.g., as defined in [15]; and (2) having the states and the deltas be elements of a join-semilattice. Causal consistency is achieved by using a causal anti entropy algorithm, e.g., one that only delivers a delta once all deltas causally preceding it have been delivered.

As previously noted, an important property for our JSON CRDT is bounding the stored metadata. In our case, the state is the pair (m, c), where m stores the state of the underlying document and c is the causal context. We now analyze the metadata overhead.

We first consider the steady state, i.e., after all conflicts have been resolved and the maximal updates are not concurrent to any others. In this case, as explained in Section 3.3, the causal context can be efficiently stored in a map from node identifiers to integers, therefore its size is $O(n \log n)$, where n is the number of replicas.. The ORMap does not store any dots, therefore comes with no metadata overhead. The MVReg stores a dot per value, therefore has a metadata size of $O(1)$ per value. An element in the ORArray has 1 dot as the *uid*, and 2 dots for the position, therefore has a size of $O(1)$ per array element. Therefore, the metadata stored for a document with D registers and array elements is $O(D + n \log n)$.

We now analyze the metadata in the presence of k concurrent operations. The worst case is for $k/2$ operations to be APPLY methods to the same array element (creating $k/2$ new roots), followed by all replicas receiving all deltas, and then $k/2$ operations to be MOVE methods to that same element (creating $k/2$ children for each root). In this case, the metadata for the position has $O(k^2)$ dots. As we use a causal anti-entropy algorithm, causal context remains $O(n \log n)$. Therefore, in this case the metadata stored for a document of size D is bounded by $O(k^2 D + n \log n)$.

7 IMPLEMENTATION

We implemented DSON in around 2500 lines of JavaScript code[37]. We expose 5 methods from the external API of Automerge [2]:

init() initializes a new empty document.
from(*obj*) creates a new document and populates it with the contents of a given JSON object *obj*.
change(*doc*, *f*): applies function f to document *doc* locally as well as locally storing the delta to be broadcast.
applyChanges(*doc*, *delta*): merges delta *delta* with the local document *doc*.
getChanges(*doc*): coalesces all deltas generated for a document since the previous getChanges operation for sending to another replica.

In a nutshell, change(doc, f) returns the delta X^δ that represents the alteration of document *doc* by function f, and applyChanges($doc, delta$) computes $X \sqcup X^\delta$ where X is the state of document *doc*, as defined in previous sections. Separate libraries implement disk persistence and network communication, e.g., encryption, authentication, access control, and implementation of an anti-entropy algorithm.

To represent a JSON object, the top level of DSON is a map. Recall that ORMaps and ORArrays recursively apply changes via a delta mutator. These recursive functions can be complex to write and reason about. To make the interaction with DSON more accessible, we follow a path similar to Automerge [2], utilizing JavaScript proxies [8] to allow a user to interact with DSON using the standard JavaScript programming model. To that end, we implemented a mechanism that transforms these operations into delta mutators and applies them. For example, figure 2 depicts the update $doc.k_1[0] = \{k_2 : 1\}$, we first create a delta mutator representing the left hand side to identify the path in the document to which this mutator is to be applied. Then the value is created by creating an MVReg for the literal and a new ORMap is created into which the MVReg is inserted under the key k_2.

We also implemented the wrapper described in Section 5.2, presented in Algorithm 4. This wrapper enables programmatic access to arrays using JavaScript indices (integers), while internally array positions are encoded using stable identifiers as defined in Logoot [44] and LSEQ [30]. Note that the code altering the state is

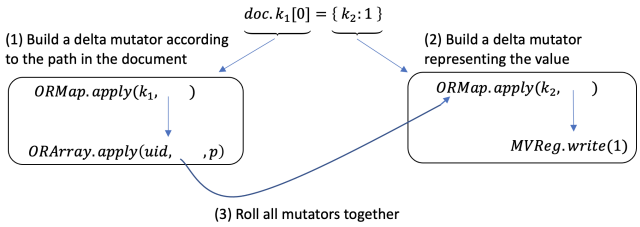


Figure 2: Transform a JavaScript operation $doc.k_1[0] = \{k_2 : 1\}$ to a delta mutation in the JSON CRDT, where p = GetPosFromIdx(0) and uid = GetUidFromIdx(0).

decoupled from the state itself, so that any choice of stable identifiers can be plugged in. The function GetUidFromIdx(idx, (m, c)) (GetPosFromIdx(idx, (m, c))) is implemented by sorting the array and returning the unique identifier (position) of the element at the index idx.

Concurrent writes support both *value* and *type* conflicts. A value conflict is where the same variable is concurrently set to different values, e.g., replica 1 sets $user.last\_access$ = 14:49 and replica 2 sets $user.last\_access$ = 14:52. This is handled by the MVReg.

Type conflicts arise because JSON objects are weakly typed. The type of a variable can change over time, and variables can be set concurrently to different types, for example, replica 1 may set $a = 1$, while replica 2 may concurrently set $a = \{\}$. The resulting variable is not defined by a singe type. The challenge is that our δ-based CRDTs do not support joins across dot stores, e.g., a join between a *DotFun* and a *DotMap* is undefined. To circumvent this, we nest each variable in an ORMap containing three keys, one corresponding to a map, one to an array and one to a value, and arbitrarily choose the following order $Map > Array > MVReg$. A type change is implemented by adding an element to this triple, and removes the other keys. The CRDT eventually converges to a single type, although values that were written with previous types remain until they are overwritten, which can be arbitrarily long.

8 EVALUATION

We compare DSON to two open-source libraries implementing JSON CRDTs: Automerge [2] and Yjs [12]. Both of these libraries optimize for collaborative text editing, but nevertheless have full support for JSON CRDTs. We note that while these libraries optimize for different behavior, they are the only CRDT libraries known to us providing full JSON datatype. Riak [11] does not provide an array data type, therefore we do not compare to it. The goal of the evaluation section is to see how the metadata size varies with the number of operations. Unlike Automerge and Yjs, our code is a research prototype and has not been optimized for performance, so we emphasize overall trends. We ran all tests on a 4-core Intel i7-8565U laptop. All documents are initialized empty, with no warmup done. Measuring document size was done using the byte size of the latest encoding provided by each library. All benchmarks are available together with the DSON source code [37].

We evaluate the map and array separately, and first evaluate the map. In our first benchmark ,we update the same key in a map consecutively with different values. The document size remains

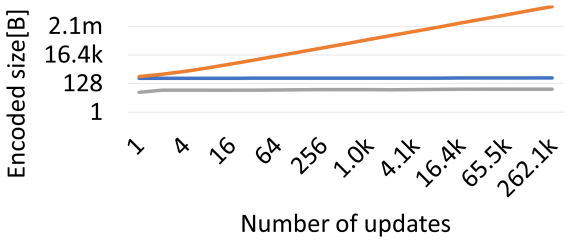
Algorithm 4 Wrapper for ORArray.
IDX represents a JavaScript index (integer), UID represents a dot, and POS represents a stable identifier.

```
1:  Shared state (m, c) with Algorithm 3
2:  Function GetPosFromIdx(idx, (m, c)) ▷ Converts index idx to stable position identifier p ∈ P.
3:  Function GetUidFromIdx(idx, (m, c)) ▷ Returns the uid of the element at index idx.
4:  procedure INSERT(idx, o_i^δ)
5:      p ← GetPosFromIdx(idx, (m,c))
6:      uid = next_i(c)
7:      return APPLY_i(uid, p, o_i^δ)
8:  procedure UPDATE(idx, o_i^δ)
9:      d ← next_i(c)
10:     p ← GetPosFromIdx(idx, (m,c))
11:     uid = GetUidFromIdx(idx)
12:     return APPLY_i(uid, p, o_i^δ)
13: procedure MOVE(old_idx, new_idx)
14:     uid ← GetUidFromIdx(old_idx, (m,c))
15:     p ← GetPosFromIdx(new_idx, (m,c))
16:     return MOVE_i(uid, p)
17: procedure DELETE(idx)
18:     uid ← GetUidFromIdx(idx, state)
19:     return DELETE_i(uid)
```

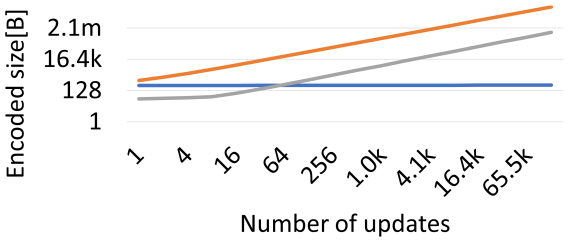
constant (it remains a single key value pair) because we do not add keys. The results are presented in Figure 3a. DSON and Yjs both maintain constant metadata size, whereas the metadata in Automerge grows linearly with the number of operations.

Next, we continuously insert and subsequently delete a key in a map. In this case the document is empty after every insert-delete pair. Figure 3b presents the results. In this workload, DSON maintains constant metadata size, whereas the metadata in both Automerge and Yjs is unbounded. The Yjs behavior is a result of its garbage collector - it creates a local object for every key which it can only clean up after its parent object has been deleted, to avoid sync conflicts [13]. Yjs attempts to garbage collect more efficiently for certain patterns but a complete and efficient solution is inherently difficult. Our delta based approach avoids the need for garbage collection altogether.

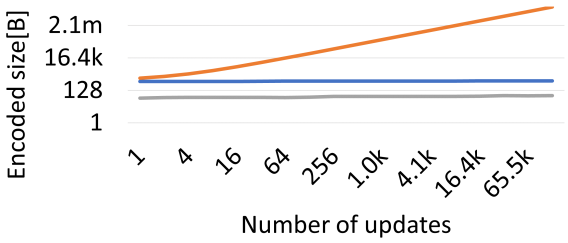
We now evaluate the array. We first measure an update only workload, i.e., the same index is consecutively updated. The reuslts are shown in Figure 3c. As expected, the DSON's metadata size remains constant. This is also the case when continuously inserting and removing array elements, as shown in Figure 3d. In the latter case Yjs also achieves constant metadata, because YATA [31] handles arrays efficiently. Note, however, that Yjs is optimized for character insertions and deletions. When inserting and removing a complex data type, the optimizations break down and the metadata grows. Figure 3e presents the metadata growth as a function of consecutively inserting and removing a map in an array, and Figure 3f presents the metadata growth as a function of inserting and removing an array in an array. While DSON keeps constant metadata, that of Yjs grows.



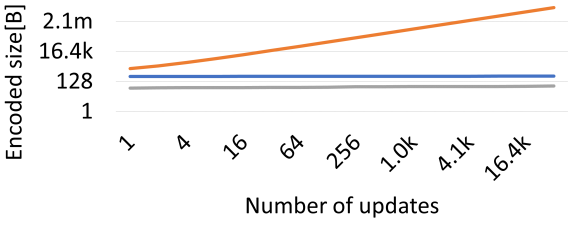
(a) Map size when updating the same key in a map.



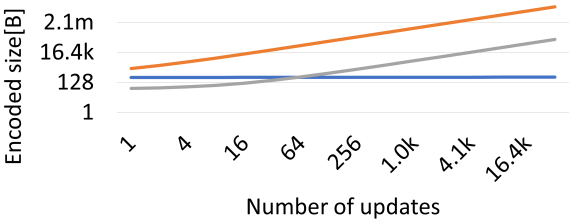
(b) Map size when inserting and removing a map key.



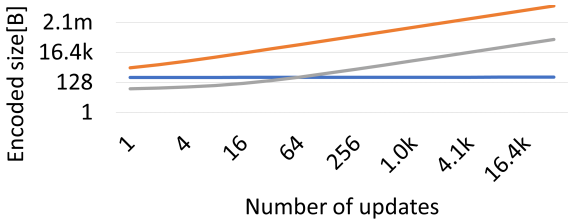
(c) Array size when consecutively updating the same index in an array.



(d) Array size when consecutively inserting and removing a character in an array.



(e) Array size when consecutively inserting and removing a map in an array.



(f) Array size when consecutively inserting and removing an array in an array.

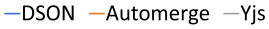


Figure 3: Micro benchmarks measuring map and array sizes.

In Section 6 we outlined our worst case metadata growth as $O(k^2)$ for k concurrent operations. This stems from how we store the position of an array element. To achieve this worst case growth, the k operations need to concurrently execute an APPLY to the same element, then deliver each other's messages, then execute MOVE operations to that same element concurrently, and finally deliver each other's messages. In this execution, the position of the element has k roots, each with k children, resulting in k^2 dots, giving $O(k^2)$ metadata growth.

To show the worst-case behavior, all replicas update every array element, then all replicas deliver each other's messages, and then all replicas sort the array. The results are presented in Figure 4. In this case, the empirical analysis matches the theoretical expectation. We argue that this scenario is rare, and under better conditions the metadata grows significantly slower than in the worst case

To show this, we implement the SORT operation. Sorting an array is a common and useful operation, and is efficiently done with our MOVE operation. As explained in Section 5, the array values are not copied at all, rather only their position changes. Furthermore, sorting can be concurrently done by different replicas, which sort the array into similar states.

We use a randomized micro-benchmark with 5 replicas to capture as many scenarios as possible. The benchmark is defined by two parameters, an update probability p_u and a sync probability p_s. Each run executes 200 steps, where in every step a randomly chosen replica performs an UPDATE with probability p_u, and a SORT with probability $1 - p_u$. After every step, the replicas sync with probability p_s. Each run is repeated 10 times, where every run measures the maximum document size within the run. The result of the micro-benchmark is the average of the maximums, with error bars depicting the standard deviation. The results are presented in Figure 5. The worst case memory size is taken from Figure 4, and the best case is measured when the array is created, before any concurrent operations occur, for the appropriate number of replicas. The analysis shows that the document size in these tests remains far from the worst-case scenario.

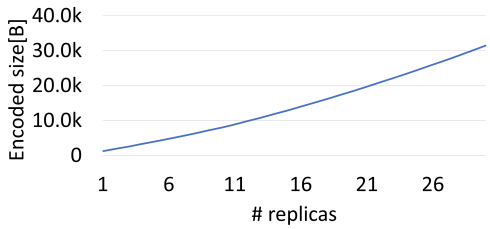


Figure 4: Document size where all replicas update all elements of an array, then sync, and then sort the array, as a function of the number of replicas.

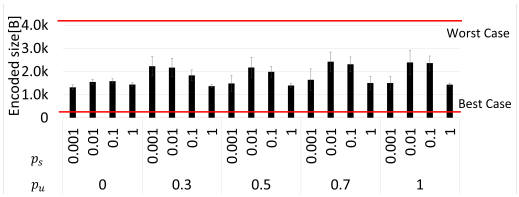


Figure 5: Document size for 5 replicas, where each replica either performs an update with probability p_u, sorts with probability $1 - p_u$, and synchronizes with probability p_s.

9 FUTURE WORK

9.1 Performance Optimizations

Enes et al. have shown that practical anti-entropy algorithms are wasteful, and in the end δ-based CRDTs perform no better than the state based solution [20]. They identify the concept of *join decomposition* which can be used to obtain optimal deltas, and show how this concept is applied to existing δ-based CRDTs. Future work is needed to construct the join decomposition for the δ-based CRDTs mentioned in this paper.

9.2 Ad-hoc Clients

We have shown that the size of the causal context is $O(n \log n)$ where n is the number of replicas. For a system with a bounded number of long-lived replicas this overhead remains small. For example, when deploying distributed document stores as a global cloud service, one could provide strong consistency within a region [7], and use DSON to coordinate updates globally. Each region would be considered a single replica, and the number of replicas is bounded and fairly static. This requires clients to be connected to some replica. For systems with an unbounded number of short-lived replicas this overhead may become significant, since any replica that connects and executes operations leaves a permanent footprint.

We now differentiate between long-lived replicas, which typically represent servers connecting the network, and short-lived replicas which represent ad-hoc, possibly disconnected, clients. For simplicity, we refer to the latter as *clients*. We propose a solution which supports an unbounded number of clients while maintaining low metadata overheads.

To this end, we note that the purpose of the dot is to uniquely identify an operation. Moreover, dots needn't dictate a causal order – up until now given two dots (c, i) and $(c, i+k)$ for some $k \geq 0$ we could say that the operation that generated (c, i) causally precedes the operation that generated $(c, i + k)$. However, this is not an inherent requirement, as the causal context is used to maintain the causal order.

We propose using a mixture of local and global dots. A client c is connected to a single replica r, and runs the same algorithm as for mutating the JSON CRDT. When the client generates a dot, it considers itself a replica and does so normally, generating a dot (c, i) for some integer i. When the client propagates its delta, it passes along a list of its locally generated dots contained in the delta $(c, i), (c, i+1), \ldots, (c, i+k)$. When the server receives the delta and list, for every client dot in the list it generates its dot, mapping $(c, i+j)$ to $(r, l+j)$ for all $1 \leq j \leq k$, where (r, l) would be the next dot generated right before the delta is received.

After generating the mapping, the replica scans the delta and alters all client dots to the corresponding replica dots. It then sends the mapping back to client c, and propagates the transformed delta to all other clients and replicas. This method allows the clients to operate as if they were replicas, while maintaining a small causal context. One drawback is if the mapping sent back from replica r to client c is lost or delayed for long enough, the client may assume the replica crashed. In such a case it may send its delta to a different replica r', which will reapply the changes. In such a case we may have a double commit of an update done by the client.

9.3 Efficient Replica Removal

The CRDT approach allows replicas to be added to the network dynamically at no additional cost, as this case is equivalent to the replica joining from the start without sending messages. Efficiently removing replicas from the network is more challenging. The changes made by a replica are retained in the casual context, which leads to an increase in metadata over time. Simply removing the relevant dots from the causal context may harm causality, since it may lead to unwanted re-application of updates. A safe method of cleaning up metadata is left for further work.

10 CONCLUSIONS

We presented DSON, a space efficient δ-based CRDT approach for distributed JSON document stores, enabling high availability at a global scale. We formally defined DSON's semantics, and proved its correctness and convergence, providing consistency guarantees which relieve application developers of the burden of reasoning about complex distributed application logic. The amount of metadata DSON stores is not dependent on the number of document updates, unlike previous approaches. We evaluated DSON empirically in this respect and showed that the metadata size stays constant as expected while varying the number of document updates, unlike the case for Automerge and Yjs which are optimized for collaborative document editing. DSON is a sound theoretical basis for efficient and robust highly available distributed document stores, which we believe can lead to efficient systems in practice.

ACKNOWLEDGMENTS

We would like to thank Dolev Adas, Ofer Biran, Michael Factor, Ronen Kat, and Adam Kocoloski for their invaluable input.

REFERENCES

[1] 2021. Akka. https://akka.io/.
[2] 2021. Automerge. https://github.com/automerge/automerge.
[3] 2021. Couchbase. https://www.couchbase.com/.
[4] 2021. Couchbase Availability. https://docs.couchbase.com/server/5.5/understanding-couchbase/clusters-and-availability/replication-architecture.html.
[5] 2021. CouchDB. https://couchdb.apache.org/.
[6] 2021. DynamoDB. https://aws.amazon.com/dynamodb/.
[7] 2021. IBM Cloudant on Transaction Engine Documentation. https://cloud.ibm.com/docs/Cloudant?topic=Cloudant-overview-te.
[8] 2021. JavaScript Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.
[9] 2021. JSON. https://www.json.org/json-en.html.
[10] 2021. Peer-base/JS-delta-crdts: Delta State-based CRDTs in Javascript. https://github.com/peer-base/js-delta-crdts
[11] 2021. Riak. https://riak.com/index.html.
[12] 2021. Yjs. https://github.com/yjs/yjs.
[13] 2021. Yjs Forum: Map Metadata Overhead. https://discuss.yjs.dev/t/map-metadata-overhead/492.
[14] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*. Springer, 62–76.
[15] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (2018), 162–173.
[16] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide, chapter 17: Conflict Management.* " O'Reilly Media, Inc.".
[17] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2014. Making operation-based CRDTs operation-based. In *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 126–140.
[18] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).
[19] Ivan Chajda, Radomír Halaš, and Jan Kühr. 2007. *Semilattice structures*. Vol. 30. Heldermann Lemgo.
[20] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient synchronization of state-based CRDTs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 148–159.
[21] Pascal Grosch, Roman Krafft, Marcel Wölki, and Annette Bieniusa. 2020. AutoCouch: A JSON CRDT Framework. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2020)*. ACM, Article 6. https://doi.org/10.1145/3380787.3393679
[22] Pat Helland. 2012. Idempotence Is Not a Medical Condition: An essential property for reliable systems. *Queue* 10, 4 (2012), 30–46.
[23] Andrew Jeffery, Heidi Howard, and Richard Mortier. 2021. Rearchitecting Kubernetes for the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. 7–12.
[24] 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.
[25] 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
[26] Rusty Klophaus. 2010. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*. 1–1.
[27] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.
[28] Lars Larsson, Harald Gustafsson, Cristian Klein, and Erik Elmroth. 2020. Decentralized Kubernetes Federation Control Plane. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 354–359.
[29] M Mihai Letia, N Preguica, and M Shapiro. 2009. CRDTs: Consistency without concurrency control. *RR-6956, INRIA* (2009).
[30] 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.
[31] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *19th International Conference on Supporting Group Work (GROUP 2016)*. ACM, 39–49. https://doi.org/10.1145/2957276.2957310
[32] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2005. *Real time group editors without operational transformation*. Research Report RR-5580. INRIA. https://hal.inria.fr/inria-00071240/document
[33] 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.
[34] Nuno Preguica, Joan Manuel Marques, 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.
[35] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. IEEE, 395–403. https://doi.org/10.1109/ICDCS.2009.20
[36] Arik Rinberg, Tomer Solomon, Guy Khazma, Gal Lushi, Roee Shlomo, and Paula Ta-Shma. 2021. Array CRDTs Using Delta-Mutations. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–3.
[37] Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, and Gal Lushi. 2021. DSON - JSON CRDT Using Delta-Mutations. https://github.com/crdt-ibm-research/json-delta-crdt.
[38] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (2011), 354–368.
[39] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel and Distrib. Comput.* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006
[40] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.
[41] solid IT. 2021. DB-Engines Ranking of Document Stores. https://db-engines.com/en/ranking/document+store.
[42] solid IT. 2021. DBMS popularity broken down by database model. https://db-engines.com/en/ranking_categories.
[43] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. 1994. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 140–149.
[44] 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.