C8DB: Geo-Replicated, Conflict-Free Document Database with Session Guarantees

Chetan Venkatesh, Durga Gokina, Christopher S. Meiklejohn

Macrometa

April 2, 2019

1 Introduction

Modern web and mobile applications are becoming increasingly concerned with providing low-latency, global operation at scale. To provide low-latency operations, applications typically need to be deployed in multiple datacenters (DCs), and user data replicated to each of these DCs, allowing user requests to be serviced by the nearest geographic DC. However, while reducing user perceived latency through the replication of data, application developers are now faced with an additional, even more difficult challenge: managing the consistency of multiple replicas.

First generation Content Delivery Networks (CDNs) serve to reduce user perceived latency when performing read operations against shared data. Each data item in the system is designated with a primary site where all data modifications occur and read-only replicas are maintained at every other site. These replicas are periodically refreshed on demand and cache eviction messages are used to expire the data stored at remote replicas. Updates are totally ordered by the primary: every replica in the system sees the same updates in the same order. Therefore, users observe *eventual consistency*: writes are performed at the designated primary replica and reads at other DCs will eventually return the result of the most recent write. While CDNs work well for a read dominated workload, they fail to assist the developer in situations where the workload may be write dominated.

To solve the issue of low-latency, write dominated workloads, every replica needs to be able to accept and process writes on behalf of the user. However, handling writes at each DC raises a number of difficult challenges. First, if all replicas in the system are expected to observe the same events in the same order, the system must use some form of coordination between replicas to obtain this order. In the case of geo-replication, this can be costly as (i.) the geographic distance between replicas communicating with one another will slow down the system to the speed of the slowest link and (ii.) remote replicas may be unavailable due to network partitions between two remote DCs. Second, if we forego the requirement on totally ordering all writes, concurrent writes for the same data item that occur at different DCs may *conflict* and the system will have to make a decision on how to resolve these conflicting writes.

Conflict-Free Replicated Data Types (CRDTs) are one way of dealing with automatic conflict resolution. CRDTs come in two flavors: *state-based* and *operation-based*. With state-based CRDTs, each CRDT is an abstract data type that is extended with a deterministic merge operation: when two updates are in conflict, the objects are merged using a predefined algorithm for processing the merge. Operation-based CRDTs rely on two properties: (i.) causal delivery of updates, ensuring that updates are delivered to remote replicas observing the causal order that the updates occurred in; and (ii.) the commutativity of concurrent operations, ensuring that any updates that are concurrent with respect to the causal order with result in the same outcome. State-based CRDTs have seen significant adoption in industry; whereas operation-based CRDTs have not had as much success, presumably due to the lack of industrial support for causal delivery in modern programming frameworks. However, as not all types of applications can be modeled using CRDTs and causal consistency (e.g., registers with access controlled by a mutex, bank account transfer with non-negative balance invariant [8]), for any solution to be comprehensive, it must offer strong consistency operations as well.

In this paper, we discuss the design of C8DB, a geo-replicated, conflict-free replicated document database. C8DB can operate in two modes: *single-master*, where the system provides strong consistency and serializable transactions; or *multi-master*, where the system provides session guarantees. Documents in C8DB are JSON documents that can contain either sequences or dictionaries: these documents can reference other documents in the system. Modifications to documents are modeled using operation-based CRDTs and conflicting updates are automatically resolved by using a predefined merge strategy depending on the type of modification and document. Modifications to documents are stored in a log and asynchronously propagated in causal order using reliable causal broadcast: users observe session guarantees while connected to a single DC. Users interact with the system using a custom SQL-like query language called C8QL for performing reads and writes and have the ability to perform atomic transactions. In *multi-master* mode, transactions exhibit *parallel snapshot isolation*.

Specifically, C8DB provides the following:

- Geo-replicated JSON document database that provides either strong consistency (*single-master*) or session guarantees (*multi-master*).
- Operation-based CRDT document model with garbage collection for automatic resolution of concurrent, conflicting updates.
- Transactions, either with serializable isolation (*single-master*) or parallel snapshot isolation (*multi-master*).

2 System Design

C8DB is a key-value document storage system designed to operate in hundreds of DCs. Each DC replicates a set of *databases*: each database contains one or more collections of documents. Documents are accessed using a query language that operates over collections allowing users to both read and write documents. Each document is assigned a unique *key*: this key is assigned by the system at the time of document creation.

A single *metadata database* is used to replicate information regarding which databases are replicated by which DCs. Within each DC, a cluster of nodes is responsible for storage of each *database*. Cluster membership within the DC is *strongly consistent* and managed by *Apache Zookeeper.*

Databases are *fully replicated*, but not each DC replicates all databases; therefore, the system is *partially replicated* where each collection contains metadata regarding the DCs that the collection should be replicated at.

Databases operate in one of two modes: *single-master*, where only one DC is responsible for processing read and write operations issued against the database and all other DCs that replicate the database are *passive*; or *mult-master*, where every DC replicating the database is allowed to accept and process both read and write operations. Clients are automatically connected to their nearest available DC. Read and write operations are automatically routed to the primary if the accessed database is a *single-master* database. Writes occurring within a single DC are synchronous.

In a *single-master* database, reads and writes observe *linearizability.* Writes are written to the primary but asynchronously delivered to *passive* replicas. In a *multi-master* database, read and write operations are processed at whatever replica receives the request and while clients remain connected to a single DC, observe *session consistency.*

2.1 Components

Each *database* is made up of three components: *global persistent streams*, a *document store*, and a *local persistent stream*. Each component is replicated for fault tolerance.

Global Persistent Streams. The first component is a set of *global persistent streams* that contain the unacknowledged updates for the remote DCs that replicate the same databases. There is one stream per destination replica and each collection. This system is realized using *Apache Bookkeeper* and uses a *pull-based* model, where remote replicas subscribe to events in one or more of the streams and updates are pulled in FIFO order, to the origin DC by the remote DC. When an update has been received by the remote DC, it is processed and acknowledged and removed from the stream, ensuring reliable delivery under failure.

Document Store. The second component is a *document store*, realized using *Facebook's RocksDB.* The document store is used for storage of events and materialized state related to each key in the collection. Each document, one for each key in the collection, is made up of two components: the *event log*, which contains all of the events, taken from the updates in the persistent streams related to this key; and, the materialized *state*, which is a materialized version of the events in the event log. This materialized state is purely an optimization, so readers do not need to wait for the document to be materialized from the event log for every read.

Local Persistent Stream. The final component is a *local persistent stream* that contains a real-time view over the changes occurring in the database. This system is realized using *Apache Bookkeeper*. Each item is the materialized *state* resulting from both local and incoming updates occurring at this replica – populated as the updates arrive.

2.2 Documents and Document Indexes

Documents in the system are formatted as JSON and are aggregated into collections. The system supports two types of collections: *generic* collections, where any free form JSON document can be stored in the system; or *edge* collections.

Generic collections can store two types of documents: one primitive type and two recursive types. Registers serve as a primitive type that can only be modified by assignment and may contain opaque values. Maps and sequences are recursive types: they may contain other maps, sequences or base types either by document reference or through value embedding. Maps are unordered dictionaries; sequences can be considered maps from array index to value. Document references are pointers to other keys in the store: upon materialization, Document references are recursively resolved to a final value in the materialized state.

Edge collections are for the storage of graph data. Edge collections ensure that all stored JSON documents take a particular form: they must contain both a *from* field and a *to* field that specifies the source and destination vertices in the graph, respectively. Edges may be annotated with a free-form type, denoting the type of relation – labeled edges.

2.2.1 Document Indexes

C8DB supports several types of indexes over documents:

1. *Geo-location.* If the document provides a field containing a latitude-longitude tuple, this can be used to facilitate geo-queries.
2. *Hash.* The hash index can be used to index a document based on a specific field, if that field happens to be unique.
3. *Persistent.* Persistent indexes are durably stored hash indexes – instead of being recreated on failure of a DC, they are stored durably on disk.
4. *Full-text.* Given a field to index, build a Lucene-style full-text search index across all documents in the collection.
5. *Skip-list.* The skip-list index indexes documents using a given field and provides support for efficient range queries.

Indexes are colocated on the same nodes as the documents that they are indexing: they are stored alongside the documents in *RocksDB* and mutated along with the document in a transaction ensuring all-or-nothing behavior under failure.

2.3 API

Clients interact with the system using a SQL-like query language called C8QL. C8QL operates over JSON document, allowing clients to insert, update, and select documents.

We provide an example of inserting, updating, and selecting documents below. First, we provide an example of inserting documents into the `Characters` collection.

```
INSERT {
    "name": "Ned",
    "surname": "Stark",
    "alive": true,
    "age": 41,
    "traits": ["A","H","C","N","P"]
} INTO Characters
```

If we wish to select documents from the `Characters` collection, we can do it as follows.

```
FOR c IN Characters
    RETURN c
```

This operation returns the documents in the `Characters` collection. We only get one document back, as we have only written a single document.

```
{
  "_key": "2861650",
  "_id": "Characters/2861650",
  "_rev": "_V1bzsXa---",
  "name": "Ned",
  "surname": "Stark",
  "alive": true,
  "age": 41,
  "traits": ["A","H","C","N","P"]
}
```

Finally, we can update the document using it's unique key assigned by the system.

```
UPDATE "2861650" WITH { alive: false } IN Characters
```

2.4 Updates

Clients interact with C8DB by issuing API commands against a replica in the system – usually, the geographically nearest replica.

When an update operation is received for a document, the receiving DC computes a *delta* from the current materialized state to the new state generated by the client's operation. This delta only describes the difference between the two and does not contain the unmodified fields.

To make this clear, we provide a concrete example. In this example, the state for a map at DC_1 starts with two keys: x, which points to 1, and y, which points to 2. When the user issues an operation to change y to 4 and to add the additional key z with value 3, an update is generated containing only the modified contents of the object: therefore, the update does not contain the key x. We refer to this update generated from a users update command as a *delta*.

Once a delta has been generated, the delta is then *decomposed* into a set of updates representing the change to each field. To make this clear, we demonstrate using the previous example. In the previous example, the update generated by the system for the map contained two keys, y, pointing to the value 4, and z, pointing to the value 3. The decomposition of this update creates two distinct updates, one for each key in the map, y and z.

These decomposed updates are then inserted into the *global persistent streams* for each DC that is replicating the collection containing this key.

The set of updates placed in the *global persistent streams* are represented by a set of decomposed delta updates and annotated with two logical clocks: (a.) a logical clock representing the causal dependencies that this update depends on[1]; and (b.) a logical clock representing the unique identifier for this update.

These updates are then subsequently applied to the event log for the document, in the *document store.* As previously discussed, each key in the system has a document in the document store where this document contains both an *event log* and materialized *state*. The event log in each document stores is a collection of event logs for each field; in the case of the previously discussed map, there is a single event log for each field: x, y, and z. The system atomically inserts the events into the event log and updates the materialized state for each set of updates received.

Finally, the materialized *state* is placed in the *local persistent stream* at the DC.

[1]This is effectively the *read-set* in traditional literature on causal consistency.

2.5 Replication and Consistency

As previously discussed, each DC provides streams of updates realized by *persistent streams.* If a remote DC wishes to replicate a collection, it subscribes to the streams of all of the other remote DC's that are also replicating that collection. Remote DC's pull a single group of updates at a time (representing a single write to the system and annotated with a single logical clock) and apply those updates locally, remotely acknowledging that the update has been applied before moving to the next group of updates. Updates are kept in persistent streams until either acknowledged by the remote DC, or when a 3 day window expires, which requires the DC rejoin the cluster and perform a full synchronization.

DCs receive updates from other DC's in FIFO order: updates are received over the stream in the order applied at the sender's log. Updates are timestamped using a vector clock that serves to capture causal dependencies: upon receipt of an update, the update is buffered and only applied when causal dependencies are met (similar to the approach taken by Lazy Replication [11].) Therefore, within a single session users observe causal consistency: the combination of monotonic read, monotonic write, read your writes, and writes follow reads; this is known as *session consistency*.

As clients are automatically routed to the nearest available DC, clients consistency guarantees may weaken for two reasons. First, if a client is a device that changes location – such as a mobile device – a previously access DC may no longer be the geographically closest DC during all interactions. Second, if a DC fails, a client may be routed to another DC to ensure availability. Under both of these circumstances, clients guarantees are weakened to *eventual consistency*.[2] These guarantees are weakened because if the inability to satisfy the following properties: (a.) *monotonic read*: clients may begin a new session at a DC that does not contain previously observed write; (b.) *read your writes*: clients may begin a session and not be able to observe their own writes; (c.) *monotonic write*: clients may begin a new session at a DC that does not contain previous write operations performed by the client at the previous replica; and (d.) *writes follows reads*: clients may begin a session at a new DC and write documents based on non-visible reads.

2.6 Value Convergence

Updates in the system may occur concurrently and therefore the value for a given key may diverge at an individual replica based on the delivery order of concurrent updates – the order they are placed in each DC's *event log* for the key. Therefore, a mechanism is needed to ensure deterministic value convergence without global coordination.

One mechanism for ensuring deterministic value convergence comes from the existing literature on Operation-based Conflict-Free Replicated Data Types (CRDTs). The intuition of Operation-based CRDTs is as follows: if updates are delivered to each replica in an order that respects the causal order of events and concurrent updates are commutative, then all replicas that receive the same set of updates will arrive at the same value: this is a property that has been formalized as Strong Eventual Consistency, with CRDTs being one data structure realizing this property [19].

However, difficulty still remains with the challenge of making inherently non-commutative operations, commute. We consider several examples below.

Registers. Registers, whose only operation is assignment to a value, pose an interesting challenge. While concurrent assignments to the same value do commute, concurrent assignment to two different values do not.

[2]This assumes multi-master collections, where they are replicated to and written at multiple DCs.

The existing CRDT literature discusses two possible design choices for registers: (a.) *Last-Writer-Wins:* where the object is encoded with a physical clock that the system uses for arbitration; (b.) *Multi-Value:* concurrent assignments result in both values being stored and returned to the caller as a set of objects that can be resolved by the user with a subsequent write.

In C8DB, we choose a traditional design that arbitrates the choice based on the lexicographical sorting of the nodes identifier in the update, opting for the greatest value based on this ordering.

References. References share a similar design to registers: they are registers that are restricted to only contain values that point to other keys in the system. As the concurrent assignment of registers to different values is non-commutative, the arbitration strategy used for registers must also be used for reference.

Maps. Maps pose an interesting challenge as well. Maps typically provide the ability to add and remove keys, as well as modify the value of a key. As we must ensure that concurrent modifications to the dictionary commute, we must also ensure that concurrent modifications to each key in the same map commute as well.

To demonstrate the challenges of updating maps, we consider the case of a map storing a single set. Consider a map $M = \{x \mapsto \{1\}\}$ that is concurrently modified by adding 2 to x at DC_1, represented as $M_1 = \{x \mapsto \{1,2\}\}$, and modified by adding 3 to x at DC_2, represented as $M_2 = \{x \mapsto \{1,3\}\}$. In this case, we can use the set union operation to produce the same result at each replica for the concurrent operations: $M_1 \cup M_2 = \{x \mapsto \{1,2,3\}\}$.

$$\begin{tikzcd}
& \{x \mapsto \{1,2,3\}\} & \\
\{x \mapsto \{1,2\}\} \arrow[ur, dash] & & \{x \mapsto \{1,3\}\} \arrow[ul, dash] \\
& \{x \mapsto \{1\}\} \arrow[ul, dash] \arrow[ur, dash] &
\end{tikzcd}$$

However, a more complicated case quickly arises under concurrent removals and updates. If we consider the same initial map $M = \{x \mapsto \{1\}\}$, what would happen if concurrently DC_1 removed the key x while DC_2 added 2 to the set stored at key x? What should the values converge to? What would the application developer assume the map converges to?

C8DB takes an approach that is similar to existing work on the distributed dictionary in Riak [3]: insertions of new keys are modeled as updates to a $\perp$ or nullary value, updates for keys take priority over removals for keys, and removals capture the elements being removed. To realize this in the previous case, the resulting map would resolve at all nodes as $M_1 \cup M_2 = \{x \mapsto \{2\}\}$: the removal of the key at replica a removes all contents at time of removal, including the value 1 and the update would add 2 into the set.

As dictionaries in C8DB are recursive, they may contain other maps, references, sequences and registers. We assume that the merge strategy outlined in this section is applied recursively throughout the map to achieve value convergence.

$$\begin{tikzcd}
 & \{x \mapsto \{2\}\} \arrow[dl, dash] \arrow[dr, dash] & \\
\{x \mapsto \bot\} \arrow[dr, dash] & & \{x \mapsto \{1, 2\}\} \arrow[dl, dash] \\
 & \{x \mapsto \{1\}\} &
\end{tikzcd}$$

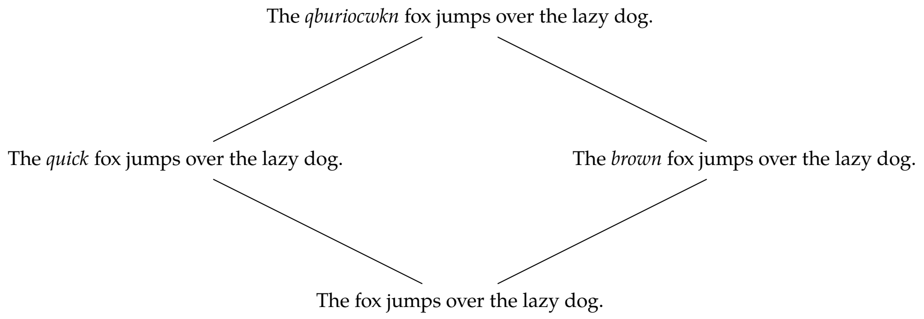
C8DB's dictionaries are recursive, therefore they can contain other dictionaries, as well as registers – using the conflict resolution semantics presented above – and sequences, discussed next.

Sequences. Sequences are also challenging as effects of modifying the sequence may not commute. We demonstrate using an example where a sequence is concurrently modified with the addition of a new word.

In this example, we show the concurrent modification of a sequence where both DC's have concurrently added a word at the same starting position. If we start with the sequence "The fox jumps over the lazy dog" and two actors concurrently modify the sequence to, at DC_1, add the word "quick" before "fox" and at DC_2, add "brown" before "fox", what does the system do? How should these updates to the same exact cell be arbitrated to ensure the correct final value?

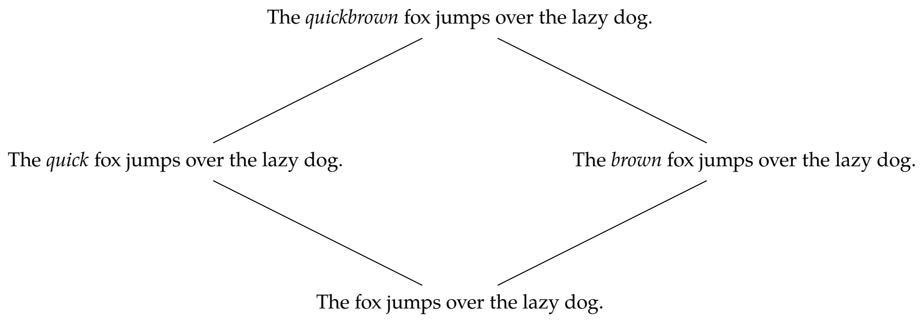
$$\begin{tikzcd}
 & \text{?} \arrow[dl, dash] \arrow[dr, dash] & \\
\text{The \textit{quick} fox jumps over the lazy dog.} \arrow[dr, dash] & & \text{The \textit{brown} fox jumps over the lazy dog.} \arrow[dl, dash] \\
 & \text{The fox jumps over the lazy dog.} &
\end{tikzcd}$$

One possible solution is to sort the identifiers of the updates at each position in the sequence lexicographically. This results in the merge of the two updates generating the following sentence "The qburiocwkn fox jumps over the lazy dog." This is far from the result that the user is expecting.


The *qburiocwkn* fox jumps over the lazy dog.
The *quick* fox jumps over the lazy dog.
The *brown* fox jumps over the lazy dog.
The fox jumps over the lazy dog.

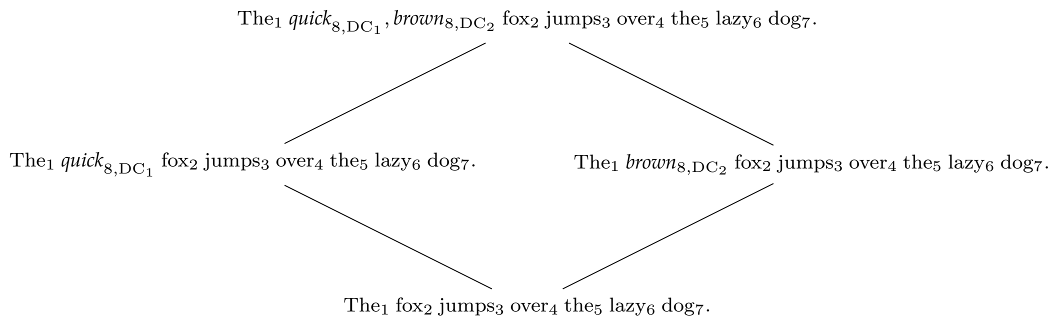
Another approach is to group the updates together – the addition of the new word, and sort the groups of updates together using lexicographical order.

In this case, we produce the final sentence after merge: "The quickbrown fox jumps over the lazy dog." This is closer to what the user would expect. However, had DC_1 and DC_2 done the opposite update, the final sentence would have been "The brownquick fox jumps over the lazy dog."


The *quickbrown* fox jumps over the lazy dog.
The *quick* fox jumps over the lazy dog.
The *brown* fox jumps over the lazy dog.
The fox jumps over the lazy dog.

C8DB take an approach to sequences as follows. In a sequence, the sequence itself is broken up into cells representing each position in the sequence. Each cell is an document reference to another document in the database – that document is written to as a register and concurrent writes to that cell are resolved using the convergence strategy for the register. Therefore, the only concurrent operation that the system must worry about is the assignment of document references to positions in the sequence. In that case, lexicographical order on the node identifier of the update is used to arbitrate the concurrent updates. This results in the final sentence: "The quick brown fox jumps over the lazy dog."

We show this below using the natural numbers as the document reference identifier for simplicity in presentation; however, in practice a randomly generated UUID is used for the key identifier and concurrent updates are detected using vector clocks.


The$_1$ *quick*$_{8,\mathrm{DC}_1}$, *brown*$_{8,\mathrm{DC}_2}$ fox$_2$ jumps$_3$ over$_4$ the$_5$ lazy$_6$ dog$_7$.
The$_1$ *quick*$_{8,\mathrm{DC}_1}$ fox$_2$ jumps$_3$ over$_4$ the$_5$ lazy$_6$ dog$_7$.
The$_1$ *brown*$_{8,\mathrm{DC}_2}$ fox$_2$ jumps$_3$ over$_4$ the$_5$ lazy$_6$ dog$_7$.
The$_1$ fox$_2$ jumps$_3$ over$_4$ the$_5$ lazy$_6$ dog$_7$.

2.7 Transactions

Transactions allow users to perform operations on groups of documents in the database. Transactions are atomic: ensuring that updates, applied against multiple documents in the DC, are applied atomically (e.g., *all-or-nothing.*)

2.7.1 Isolation Levels

Transactions in C8DB exhibit a variety of isolation levels, depending on the objects being written to and read from. For all transactions accessing shared resources, the *wait-die* algorithm is used for deadlock detection and resolution only for transactions executing in the same DC.

For databases operating in *single-master* mode, transactions are *serializable.* Transactions will be totally ordered across all replicas. Both read and write locks are explicitly taken at the start of the transaction. Concurrent transactions abort under either *read-write* or *write-write* conflicts.

For databases operating in *multi-master* mode, transactions exhibit *parallel snapshot isolation.* Both read and write locks are taken upon write or specified at the start of the transaction. Transactions are ordered related to the causal order of updates and concurrent transactions are arbitrated using the ordering strategy outlined above.

2.8 Garbage Collection

By now, it should be clear that the *event log* stored inside each document grows indefinitely. This is required to ensure that at any point, when a new update arrives, the materialized *state* for the document can be regenerated.

Recall from earlier, updates are stored as operations modifying part of a documents state. In the case of primitive types, the update sets the value of a register. In the case of recursive types, an update modifies an individual field (e.g., a specific field in the dictionary.)

Now, since updates to a field are totally ordered – updates to documents are causally ordered and a total order is arbitrated using the node identifier of the update – the system needs only to retain the most recent event for a given field.

3 Related Work

In terms of system design, Bayou [6] is the closest system to C8DB. In Bayou, intermittently connected replicas accept writes to data items locally to a log, apply those changes to a local data store, and periodically propagate the log using anti-entropy [5]. Log updates are considered tentative until sequenced by a designated primary replica (one copy serializability) when an update is finally considered committed. When the final sequence is established for a write, the log entry is discarded because the log update is only stored until sequencing, when writes might have to be re-sequenced and reapplied locally based on the final sequence. Since, writes may have to be undone and reapplied until the final sequence of writes is established, providing users with a changing view over data until commit time. Therefore, users may see alternating values if they choose to read tentatively committed data, instead of data from the final committed sequence.

Regarding differing consistency levels, Bayou popularized the notion of both eventual and session consistency [21], where users were guaranteed to see a non-decreasing set of reads and writes: monotonic read and monotonic write; and observe causality of their own writes: writes follow reads and read your writes. Clients in Bayou would choose a DC to failover to, based on a DC who met a clients read and write dependencies. Lazy replication [11] proposed the use of logical clocks for buffering writes and applying them in causal order, a technique inspired by Lamport's seminal work on clocks [12] and is seen in other causally consistency systems such as COPS [14], Eiger [15], and GentleRain [7].

Regarding the detection and resolution of conflicting writes, Grapevine [2] is notable for proposing the use of a Last-Writer-Wins strategy for resolution using replica provided timestamps. Locus proposed version vectors for identifying conflicting updates, where Coda [9], Ficus [16], and Bayou [22] used this technique for identifying conflicting operations and allowed the user to specify conflict resolution functions.

Regarding the automatic resolution of conflicting operations, operational transformation [20] is a technique for automatically resolving conflicts by ensuring all operations that modify replicated data commute. Operation-based conflict-free replicated data types [19] (CRDTs) order operations using causal consistency and provide automatic conflict resolution by ensuring all concurrent operations commute. State-based CRDTs [19] ensure automatic conflict resolution under weak ordering by propagating state, where the state must form a join-semilattice, ensuring any two states are always mergeable. Dictionary CRDTs [3] have found use in both academia and industry, whereas some dictionary designs have been based on modeling the popular JavaScript Object Notation [10] dictionary semantics. Graph CRDTs [19] have been modeled previously using CRDT-based vertex and edge sets. Several designs have been proposed for modeling a sequence with CRDTs. [17, 18, 13, 25, 24]

Regarding geo-replicated, strongly consistent systems, Spanner [4] provides geo-distributed transactions with strict serializability. Spanner keeps the cost of committing geo-distributed transactions down by assigning timestamps based on the bounds of possible error: this error is kept to a minimum through the use of atomic clocks and GPS transceivers. CockroachDB is an implementation of a similar model to Spanner, however does not rely on specialized hardware, and therefore provides slightly weaker guarantees than Spanner: serializability, instead of strict serializability. Calvin [23] provides serializability for geo-distributed transactions with low overhead by preallocating timestamps to transactions and applying updates based on timestamp order.

Regarding geo-replicated, causally consistent systems, COPS [14] first proposed tracking nearest explicit dependencies with explicit dependency checks before installing an update. To support DC failover, clients in COPS, like in Bayou, would track their latest read and write dependencies and block until a DC was available to service the client, ensuring causal consistency. COPS-RT [14] and Eiger [15] further extended this design with causally consistent read and write transactions respectively, in a manner that ensured availability by avoiding blocking write transactions – using conflict resolution functions – and providing a causally consistent read transaction that never blocked on missing dependencies – by returning a causally consistent cut of writes in snapshots. GentleRain [7] provides a design that uses a single scalar for capturing causality, improving throughput and efficiency at the cost of visibility latency. Cure [1] further refines this design by using a vector of scalars, allowing the system to make progress in the face of partitions thereby reducing visibility latency.

References

[1] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 405–414. IEEE, 2016.

[2] Andrew D Birrell, Roy Levin, Michael D Schroeder, and Roger M Needham. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[3] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 1. ACM, 2014.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[5] Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[6] Alan Demers, Karin Petersen, Mike Spreitzer, Doug Terry, Marvin Theimer, and Brent Welch. The bayou architecture: Support for data sharing among mobile users. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 2–7. IEEE, 1994.

[7] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[8] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *ACM SIGPLAN Notices*, volume 51, pages 371–384. ACM, 2016.

[9] James J Kistler and Mahadev Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, 1992.

[10] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[11] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[13] Mihai Letia, Nuno Preguiça, and Marc Shapiro. Consistency without concurrency control in large, dynamic systems. *ACM SIGOPS Operating Systems Review*, 44(2):29–34, 2010.

[14] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[15] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[16] Peter L Reiher, John S Heidemann, David Ratner, Gregory Skinner, and Gerald J Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer*, pages 183–195, 1994.

[17] Hyun-Gul Roh, Jin-Soo Kim, Joonwon Lee, and Seungryoul Maeng. Optimistic operations for replicated abstract data types. Technical report, Technical Report CS/TR-2009-318. KAIST, 2009.

[18] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[19] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[20] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM, 1998.

[21] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149. IEEE, 1994.

[22] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 172–182. ACM, 1995.

[23] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

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

[25] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE transactions on parallel and distributed systems*, 21(8):1162–1174, 2010.