CRDV: Conflict-free Replicated Data Views

Nuno Faria
nuno.f.faria@inesctec.pt
INESCTEC and University of Minho
Portugal

José Pereira
jop@di.uminho.pt
INESCTEC and University of Minho
Portugal

Abstract

There are now multiple proposals for Conflict-free Replicated Data Types (CRDTs) in SQL databases aimed at distributed systems. Some, such as ElectricSQL, provide only relational tables as convergent replicated maps, but this omits semantics that would be useful for merging updates. Others, such as Pg_crdt, provide access to a rich library of encapsulated column types. However, this puts merge and query processing outside the scope of the query optimizer and restricts the ability of an administrator to influence access paths with materialization and indexes.

Our proposal, CRDV, overcomes this challenge by using two layers implemented as SQL views: The first provides a replicated relational table from an update history, while the second implements varied and rich types on top of the replicated table. This allows the definition of merge semantics, or even entire new data types, in SQL itself, and enables global optimization of user queries together with merge operations. Therefore, it naturally extends the scope of query optimization and local transactions to operations on replicated data, can be used to reproduce the functionality of common CRDTs with simple SQL idioms, and results in better performance than alternatives.

CCS Concepts

• **Information systems → Data management systems**; **Parallel and distributed DBMSs**; **Distributed database transactions**.

Keywords

Distributed databases, geo-replication, local-first software, CRDTs

ACM Reference Format:
Nuno Faria and José Pereira. 2025. CRDV: Conflict-free Replicated Data Views. In *Proceedings of International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, Article 25, 15 pages. https://doi.org/10.1145/3709675

1 Introduction

There is a renewed interest in distributed data management techniques that minimize the coordination needed to execute queries and updates, namely, for geo-replication, where partitions otherwise lead to blocking [1, 16], and local-first software [39], that keeps data close to its owner to enforce ownership and improve performance. The scope of what is possible in a distributed data processing system without coordination has been captured by the CALM theorem [33] as *monotonicity* that, informally, means that conclusions based on partial, locally available, information will always hold.

In practice, Conflict-free Replicated Data Types (CRDTs) [76] provide a concrete programming paradigm as an object-oriented library of common abstract data types such as *sets*, *lists*, or *maps*, with restricted interfaces that can be replicated, concurrently updated, and eventually converge to the same state. In fact, some CRDT proposals are based on additional pragmatic assumptions, for example, those that restrict some concurrent operations [5].

To achieve this, CRDTs take two main approaches: state-based and operation-based [71]. The former allows each replica to update the local state, propagate it to others, and assume a merge operation that reconciles any two instances. This merge operation constitutes a semi-lattice that guarantees convergence and determinism. Alternatively, the latter approach transmits only the operations, which are partially ordered and executed to eventually obtain the same state. This reduces overhead, as the state does not have to be copied, but requires idempotency and causal delivery guarantees to ensure consistency across different replicas.

A common example showing the applicability of convergent replicated data is the online shopping cart problem [10], in which customers should be able to add/remove items from it even with network partitions. Instead of pushing divergent versions of data for the application to reconcile, the shopping cart can be modeled as an OR-Map [71] from product identifiers to quantities, ensuring that the latest version is kept even when propagation to replicas is asynchronous and that items are not inadvertently dropped with concurrent conflicting operations (i.e., *add-wins* semantics). CRDTs have also been sought in NoSQL key-value stores that lack server-side DML and multi-operation transactions to avoid lost updates: Concurrent clients read the same key and compute updated values locally, which are merged when written back [2, 8, 51, 73, 81].

However, integrating CRDTs into SQL systems is challenging [42]. First, simply using CRDTs in the stored data model while offering a query language does not ensure that the queries are actually monotonic. Second, and the one that we mainly address in this paper, encapsulated CRDTs within SQL database systems hinder the opportunities for global query optimization and impact performance.

In fact, current proposals follow one of two approaches. The first embeds CRDTs as custom data types, for instance, using PostgreSQL object-oriented extensibility [79]: A table column can thus be typed as a CRDT, such as an OR-Set. However, this puts the internal structure and operation of the CRDT – state merge or ordered execution, respectively, for state-based or op-based approaches, which can be quite complex – out of the scope of the query optimizer. Others are limited in terms of conflict-free semantics [6, 47, 89, 90], such as

Authors' Contact Information: Nuno Faria, nuno.f.faria@inesctec.pt, INESCTEC and University of Minho, Portugal; José Pereira, jop@di.uminho.pt, INESCTEC and University of Minho, Portugal.

SIGMOD '25, Berlin, Germany

https://doi.org/10.1145/3709675

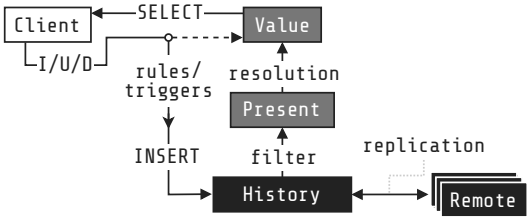


Figure 1: Conflict-free replicated data as stacked views.

Table 1: CRDV's requirements support in SQL DBMSs.

DBMS	Views	Triggers/ DDL triggers	Complex Types	Async. Rep.
PostgreSQL	Yes[31]	Yes[30]/Yes[24]	Array/JSON[27, 28]	Yes[20]
Oracle	Yes[67]	Yes[65]/Yes[66]	Array/JSON[68, 69]	Yes[64]
SQL Server	Yes[56]	Yes[52]/Yes[53]	JSON[54]	Yes[55]
MySQL	Yes[63]	Yes[(a)][62]/No	JSON[60]	Yes[61]

[(a)] Not supported on views.

providing *last-writer-wins maps* in the form of replicated relational tables. Moreover, neither allows for different merge semantics in different queries on the same data, depending on the situation, which is desirable but still an open problem [71].

In this context, our contribution is to provide an abstraction for convergent data that better fits mainstream SQL systems. In detail, this means that performance-wise, the operations within (e.g., merging) and on (e.g., user queries) convergent data are expressed in terms of relational operations and globally optimized, together, by the query engine. In terms of programming abstraction, our goal is that merge strategies can be expressed in SQL, thus not being limited to either a simple *map* or a library of encapsulated options.

The core tenet of our approach is the definition of conflict-free replicated data using layered views (Section 2). At the lowest level, we show that the key issue is the efficient representation and querying of causal relations to prune concurrent versions. Therefore, we present and later evaluate the different options available in SQL systems (Section 4). At a high level, we provide a *cookbook* that, based solely on relational data, can reproduce a range of typical CRDTs, including complex generalized nested structures (Section 3). This approach naturally and efficiently allows multiple simultaneous conflict resolution rules for the same data, as sought but not easily achieved by encapsulated CRDTs [71], and interoperability with local ACID transactions at each site. The integration with cross-site transactional guarantees is, however, not considered in this work.

As shown with concrete examples, this architecture allows global optimization across user queries, merge semantics of the top layer, and replication and causal ordering of the bottom layer. A key consequence is that, according to best practices in database systems, the introduction of redundancy (materialization and indexing) is deferred to a deployment option, thus allowing optimal handling of diverse workloads. This is, to the best of our knowledge, the first time such a possibility has been proposed for conflict-free replication. Furthermore, we compare CRDV with state-of-the-art implementations of CRDTs, namely, Riak KV, ElectricSQL, and Pg_crdt, showing that our approach offers competitive performance even with low-level implementations (Section 5).

Finally, we provide an overview of related work (Section 6) and point out interesting research directions (Section 7). All code, scripts, and results are available at https://github.com/nuno-faria/crdv.

2 The CRDV Approach

Programming convergent data in high-level SQL allows developers to easily express merge semantics and operators to achieve the best performance (e.g., by controlling materialization and indexing), while at the same time hiding the complexity of replication and taking advantage of existing infrastructure (e.g., asynchronous replication). This is achieved with the architecture shown in Figure 1 with two views, *Value* and *Present*, stacked on a replicated *History* table. The different layers thus separate different concerns, namely, the lowest (*History*) is concerned with replication, the middle (*Present*) with garbage collection, and the top (*Value*) with merge semantics. Using views, we avoid imposing a single materialization strategy and allow global optimization across layers and with user queries, in contrast to existing systems that encapsulate CRDTs as column types [79].

Write operations are captured and transformed into Inserts to *History*. This layer is the source and target of replication, eventually being copied to and from all replicas. Therefore, it plays a similar role to the write-ahead log in a transactional system with the key difference that it is partially ordered by Lamport's causality relation [44], that is, there can be concurrent unmerged writes to the same item. The *Present* layer is a generic transformation that filters out versions that no longer contribute to the final value. The generic nature means that, to support varied and rich conflict resolution rules (Section 2.4), it must provide all the latest concurrent versions to the *Value* layer, i.e., operate under *multi-value register* semantics [75]. To this end, each row is tagged with a vector clock [14] to determine causality (Section 2.2). The performance of operations on vector timestamps is thus key in CRDV. The current *Value* is computed from the causal *Present* by relying on specialized, user-defined, and often simple conflict resolution rules. In summary, CRDV follows a state-based approach [71] with row granularity, potentially having multiple versions of the same row coexisting simultaneously and being merged according to their content.

To make CRDV transparent to the application, we assume that the database system supports views for virtual representations of data, as well as triggers/rules to allow write statements to be captured and modified using custom logic. Complex types such as arrays or JSON are also desirable to model some metadata (Section 4.1). For replicating the *History* layer, we assume asynchronous replication. These features are found in all the major relational database systems, as depicted in Table 1. The main limitation is with MySQL, which does not support triggers on views, which means that writes need to call procedures explicitly.

The remainder of this section examines how these layers work in detail, by describing their schema, and how updates, queries, and replication operations are processed.

2.1 Data schema

To illustrate the CRDV approach, consider an application that uses a relational table with a primary key and some additional data. For

Listing 1: Rule to add a row to a replicated table.

```
1  CREATE RULE insert_rule AS
2    ON INSERT TO Value DO INSTEAD
3    INSERT INTO History
4    SELECT key, data, 'add', siteId(), (t).lts, (t).pts
5    FROM nextTimestamp() AS t
```

simplicity, we illustrate with the columns *key* and *data*, respectively, although the discussion applies to composite keys and multiple data columns. Other alternatives are discussed in Section 3. As such, the schema for the *Value* layer only contains the columns *key* and *data*.

Present and *History* have a different schema from *Value*, storing additional metadata. In detail, in addition to *key* and *data*, we keep a label that states whether the row has been added or removed (i.e., if it is a tombstone), named *op*. In addition, we keep an identifier for the originating site (*site*) and a logical timestamp for each operation that captures causality [44] (*lts*). These are used for replication and to ignore rows that have become obsolete. We also optionally keep additional metadata required for the desired merge semantics, for instance, a physical timestamp for *last-writer-wins* reconciliation (*pts*). It should be noted that *key* is not unique in these layers, as multiple concurrent versions can be stored.

2.2 Writing data

Modifications to the *Value* layer are redirected as Inserts of new versions in the *History* layer (Figure 1). These Inserts take the provided data and complement them with metadata. Namely, whether it represents a new value or removal (*op*), the site's identifier (*site*), the logical timestamp (*lts*), and, optionally, the physical timestamp (*pts*). Each row inserted in the *History* layer is expected to eventually be replicated to the same layer in all other replicas.

The key aspect of the *History* layer is managing the partial order that might result from uncoordinated concurrent updates. For this, a logical timestamp represented using a vector clock [14] is used, where each element refers to the latest logical time seen at that site when the row was created. While determining causality is not required in state-based replication, it is needed to enable rich conflict resolution rules in CRDV (Section 2.4). The timestamp is computed by querying the *Present* layer for the current maximums and then incrementing it for the local site, e.g., $[7, 4, 1]$ at site 0 advances to $[8, 4, 1]$. This provides correct timestamps regardless of the materialization strategy and avoids the overhead and potential concurrency bottleneck of having a separate clock table.

Listing 1 shows a SQL rule performing the redirection of an Insert. In detail, Inserts to the *Value* layer (line 2) are translated into Inserts to the *History* layer (line 3), with the respective data (lines 4 and 5). Updates and Deletes follow a similar strategy.

2.3 Filtering

Logically, the *Present* layer is a view of the *History* table that filters only the rows in the causal present, i.e., that contribute to the visible state, hiding much of the complexity of conflict-free replication. A version v_1 with logical time lts_1 is replaced by a version v_2 with logical time lts_2 if both are identified by the same key and $lts_1 < lts_2$, which is defined as $lts_1[i] < lts_2[i], \forall i \in [0, nSites())$. In other words, if v_1 happens before v_2 ($v_1 < v_2$). If $v_1 \not< v_2 \wedge v_1 \not> v_2$, the versions are concurrent and, consequently, are both considered ($v_1 || v_2$). Listing 2 illustrates a definition of the *Present* layer. Briefly, we consider all versions (lines 2, 3) that were not causally replaced (lines 4-9). We refer to this strategy of constructing the *Present* layer solely by querying the data in the *History* layer as *no-mat*.

Listing 2: Definition of the *Present* view in the *no-mat* strategy. Note that this is a straightforward representation in SQL and not necessarily optimized. *vclock_lte* computes '$\leq$' of two logical timestamps.

```
1  CREATE VIEW Present AS
2    SELECT *
3    FROM History t1
4    WHERE NOT EXISTS (
5      SELECT 1
6      FROM History t2
7      WHERE t1.key = t2.key
8        AND t1.lts <> t2.lts AND vclock_lte(t1.lts, t2.lts)
9    )
```

However, it is wasteful to define *Present* strictly as a view, as we expect only a small subset of *History* to affect the result. Therefore, we seek to materialize it and devise an incremental maintenance procedure. In Section 4.2, we describe how the same procedure also removes data from *History*, avoiding unbounded growth.

Full materialization of Present (sync). In this case, *Present* is defined as a table and is synchronously updated when a new local version is created, reducing the read complexity. This is done with a trigger on *History* executed within the transaction that Inserts the new row. When materializing *Present*, this trigger Deletes from it the versions that will now become obsolete (if any) and subsequently Inserts the new version.

Partial materialization of Present (async). In some cases, the synchronous materialization of *Present* may be undesirable, for instance, due to its interaction with local isolation (Section 2.5). In this case, *Present* is defined as a view that filters and combines new rows from *History*, as in Listing 2, but using existing materialization. A periodic procedure performs asynchronously the same actions as the full materialization performs synchronously. This process is analogous to the data storage strategy commonly found in Hybrid Transactional Analytical Processing (HTAP) systems [70].

In short, the *async* approach should result in a faster response time for writes, as the filtering operation is avoided. In contrast, reads will have to consider both *Present* and *History* data to ensure *read-your-writes* semantics. Choosing one alternative over the other will depend on the type of workload being executed (Section 5.3.1).

2.4 Reading data

Applications in CRDV read by issuing Select statements on the *Value* layer, which exposes the application schema without metadata. The key issue here is how concurrent versions in the causal present are reconciled, including resolving conflicts.

Common approaches for collections where the same key can be added and removed concurrently are *add-wins* (*AW*) or *remove-wins* (*RW*). For conflicting additions on the same *key*, one option is to return all conflicting values. This option, known as *multi-value register* (*MVR*), defers the resolution to the application. However, doing so at the application level increases development complexity

and means that different applications might need to reimplement the same rules. Another common approach is the *last-writer-wins* (*LWW*) rule, which keeps the last written value according to a physical timestamp. Although simple to implement, it might not be acceptable depending on the requirements. Rules can also be defined on the basis of the conflicting values themselves. For instance, *average* (*Avg*), *maximum* (*Max*), or *minimum* (*Min*) for numerical values, and *enable-wins* (*EW*) or *disable-wins* (*DW*) for flags.

One property of CRDV is that, precisely because conflict resolution rules are data processing operations, they can be expressed directly with the query language. This provides a universal runtime implementation for all applications and enables the query planner to optimize these rules. The output format can also be defined in the query language, e.g., native row format or JSON.

Furthermore, this allows us to have multiple simultaneous concurrency semantics depending on the situation, which was until now an open problem in CRDTs [71]. Figure 2 illustrates multiple conflict resolution rules applied to a table with integer *data* (Figure 2a). The data are shown in the *Present* layer, which selects the latest rows from *History* based on the respective vector clocks. Thus, multiple rows with the same key have concurrent timestamps. There is an *add* to $k1$, a *rmv* of $k2$, two concurrent *add* and *rmv* to $k3$, and two concurrent *adds* to $k4$. The *LWW* view (Figure 2b) uses a window function to rank different versions of the same *key* by their physical timestamp (lines 4-6) and select the most recent (line 9). This means that $k3$ is excluded from the result, as the *rmv* is newer, while for $k4$ the value considered is 40. The *AW+MVR* rule (Figure 2c) excludes *rmvs* (line 5) and aggregates the remaining values into an array (lines 2, 3, 6). Thus, $k4$ is $[4, 40]$. The *AW+Avg* rule (Figure 2d) projects the average value for elements with concurrent *adds* (lines 2, 5). As a result, $k4$ will be projected as $\frac{4+40}{2} = 22$.

One thing to note is that we can create views over existing views. For example, to improve code reusability in Figure 2, we can create a view with the base *add-wins* semantics and build the *MVR* and *Avg* rules over it. We can also expect good performance when doing so, as the planner can optimize the query as a whole (Section 5.3.2).

(a) *Present* data

key	data	op	site	lts	pts
k1	1	add	1	[10, 1]	(122185,1)
k2		rmv	1	[13, 1]	(123447,1)
k3	3	add	1	[14, 1]	(123720,1)
k3		rmv	2	[9, 2]	(131233,1)
k4	4	add	1	[15, 1]	(124948,1)
k4	40	add	2	[9, 3]	(132198,1)

(b) *Value* view for *LWW*

```
1 CREATE VIEW ValueLww AS
2   SELECT key, data
3   FROM (
4     SELECT key, data, op, rank() OVER (
5       PARTITION BY key
6       ORDER BY pts DESC, site)
7     FROM Present
8   ) t
9   WHERE rank = 1 AND op = 'add'
```

key	data
k1	1
k4	40

(c) *Value* view for *AW + MVR*

```
1 CREATE VIEW ValueAwMvr AS
2   SELECT key, array_agg(
3     data ORDER BY data) data
4   FROM Present
5   WHERE op = 'add'
6   GROUP BY key
```

key	data
k1	[1]
k3	[3]
k4	[4, 40]

(d) *Value* view for *AW + Avg*

```
1 CREATE VIEW ValueAwAvg AS
2   SELECT key, avg(data) data
3   FROM Present
4   WHERE op = 'add'
5   GROUP BY key
```

key	data
k1	1
k3	3
k4	22

Figure 2: Example of different conflict resolution rules applied to a table with integer *data*.

2.5 Transactions

As the proposed approach is built on the client interface of a transactional system, it allows a natural integration with existing ACID concurrency control and recovery mechanisms. The key principle is that changes to CRDV tables made in a transaction are also visible to a remote site as a transaction. However, eventual consistency does mean that transactions from different sites might be made visible in different orders at different destinations.

If the *Present* layer is fully materialized (*sync*), then operations at the same site will implicitly be isolated according to the local concurrency control mechanisms. In contrast, the *async* approach allows two local transactions that modify the same item to commit concurrently, since they only Insert new rows into *History*, which does not trigger conflicts. If we want to ensure that this mode also follows local transactional semantics, we can keep a separate table with the relevant keys for the stored items and modify the rule that diverts writes to *History* to force the conflict on rows in such a table, by modifying it or using Select ... For Update. Alternatively, if available, we can also use explicit locking functions [18].

While CRDV operates on conflict-free data, its semantics can be extended to support cross-site transactional guarantees. Very briefly, write-sets can be collected with triggers and sent to a leader site to be certified [11], replicated, and completed when acknowledgments have been received from a majority of sites. The main challenge concerns how the system should act when strong and eventual transactions are executed on the same data. We believe that CRDV may also be beneficial in this context, for instance, with resolution rules such as *strong-wins*. However, the complexity of this issue means that this topic cannot be addressed in this work, thus we consider it an interesting future research direction (Section 7).

2.6 Schema Changes

The asynchronous logical replication offered by relational database systems often does not support replicating DDL statements [23]. However, triggers on DDL statements are commonly available (Table 1), allowing us to capture, process and replicate schema changes to remote sites, similarly to how regular writes are handled.

The asynchronous nature of CRDV means that schema changes cannot be arbitrarily accepted, as inconsistencies may emerge. For instance, dropping a column in one site while writing to it in another. Solutions that have addressed this issue, such as F1[72], can be used. Briefly, an unsafe schema change – e.g., dropping a column – can be converted into two safe ones – deactivating it and then removing it. Only after the first change has been applied in all sites – e.g., using a *counter* to keep track – can the second one be executed.

A non-relational approach to this problem is to encode the schema directly in the data (Section 3.3). Although CRDV already handles conflicting changes in this case, this solution requires the user to be knowledgeable about how the data are organized.

3 The CRDV Cookbook

The CRDV architecture assumes that the design of the relational schema and the conflict resolution is done by developers so that it captures the specifics of each application. In this section, we show that this is feasible by describing how to deal with typical features of relational schemas and how to reproduce common CRDTs.

3.1 Foreign keys

The use of foreign keys to link entities in relational data is directly compatible with CRDV by making a data column refer to the key, or part of the key, of another entity. However, this raises the issue of referential integrity when merging concurrent updates [7]. As an example, consider that B is referenced by A by key k. B could be updated while concurrently k is removed from A, for example, A represents courses, B represents students in course k, and a student is added to B while concurrently the course k is removed from A. Although this represents a conflict, the actual operations are fundamentally done on separate structures, and thus the outcome is always "B stopped being referenced by A".

Depending on the requirements, we might want to ensure that B is still referenced in these cases. In CRDV, this is achieved by marking k as *added* in A in the same transaction where B is updated, effectively entangling both structures. This can be done manually or with a trigger that runs on updates to B. With this, the concurrency rules used on A (e.g., AW or RW) will dictate whether or not B remains visible. Similarly, cascading deletions can be expressed by marking the referenced row as deleted, as appropriate.

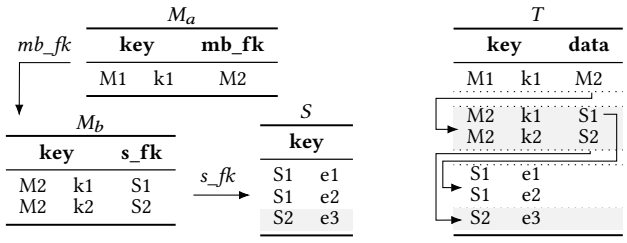
3.2 Reproducing CRDTs

CRDTs encapsulate data representation and merge semantics. To show that these can be expressed with CRDV, we implement common types such as [76]: *registers*, as single opaque values; *sets*, unordered collections of unique values; *maps*, unordered collections of key-value pairs; *lists*, ordered collections of values; and *counters*, numeric values that can be incremented or decremented.

3.2.1 Registers. A *register* can be modeled as a single-row table, hence, without a key. Concurrent updates result in conflicting versions that can be reconciled with *MVR*, *LWW*, *EW*, and so on. It is likely that *registers* are found within *maps*, as described below.

3.2.2 Maps and sets. A *map* is a table with the appropriate key and value, while a *set* is a table without data columns, only tracking if the keys are present or have been removed. In both cases, we can use LWW to resolve conflicts. In the case of *sets*, AW and RW are also straightforward, but with *maps* they require that the values of conflicting updates be themselves reconciled, e.g., using *MVR*.

3.2.3 Lists. *Lists* are modeled using the *key* as the index on which the elements are sorted. The indexes must express a consistent linear order across all sites. Taking into account two indexes l_i and l_j, we must ensure that 1) $l_i \neq l_j$, to allow insertions between l_i and l_j, and 2) if $l_i < l_j$ in one site, then $l_i < l_j$ in all sites.



(a) Relational approach **(b) Generic approach**

Figure 3: Example of a nested structure implemented with two distinct approaches. Metadata columns have been omitted.

We base our index generation on the LSEQ algorithm [58], where indexes are encoded using strings and each character is analogous to a level in a trie. The generation algorithm supports three variations: optimized for appends, for prepends, and for random inserts. To guarantee that each index is globally unique, we also append the local site's identifier, which in turn precludes causal conflicts.

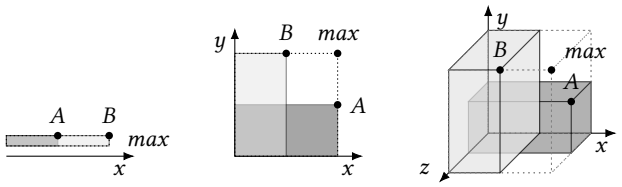
3.2.4 Counters. To model *counters* in CRDV, we consider a table with *key* and *data* columns. For incrementing or decrementing, a new version is added using, e.g., the site's id as *key*, which precludes intersite conflicts. The *data* column reflects the previous value of that *key* (zero if not present) plus the new delta. For reading, a sum is computed over the partial values. This strategy is based on value-splitting techniques [12, 57], and comes with the added advantage of enabling *bounded counters* [5] to be easily implemented.

3.3 Nested structures

In CRDV, nested structures, i.e., data structures comprised by other structures, can be implemented with varying degrees of abstraction. At the lowest level, different entities can be modeled by different tables, with links explicitly established by foreign keys. Figure 3a illustrates this approach with three entities, M_a, M_b, and S, with columns *mb_fk* and *s_fk* linking M_a to M_b and M_b to S, respectively. At the highest level of abstraction, all structures can be encoded into a single generic table, with the *data* column implicitly linking different entities. Figure 3b shows an example of a *map* structure $M1$ containing a key $k1$, whose *data* refers to *map* $M2$. In turn, the two entries of $M2$ refer to *sets* $S1$ and $S2$, respectively. Thus, $M1 = \{k1 : \{k1 : \{e1, e2\},\ k2 : \{e3\}\}\}$. Finally, a hybrid approach can store different data types in distinct tables, but allow different entities of the same type to be stored together. In Figure 3's example, entities M_a and M_b would be stored together in a *Map* table.

In all cases, the complete structure is materialized by joining the respective conflict resolution views. Since the *key* is indexed, we expect the same performance in all approaches, even though the generic approach relies on self-joins. To coax the planner to always use the index, we rely on Lateral joins [26], resulting in good performance independently of the number of joins (Section 5.3.3).

While the generic approach makes it easier to change the schema, it expects the user to know how the data are stored in the generic table. Furthermore, storing different types in the same table leads to null attributes, e.g., *data* in *sets*. Thus, we consider the relational approach to be the optimal way to model nested structures in CRDV.



(a) $N = 1$ $A = [1], B = [2]$ **(b)** $N = 2$ $A = [2, 1], B = [1, 2]$ **(c)** $N = 3$ $A = [2, 1, 1], B = [1, 2, 2]$

Figure 4: Geometric representation of several logical timestamps in N dimensions. *max* is the pointwise maximum of A and B.

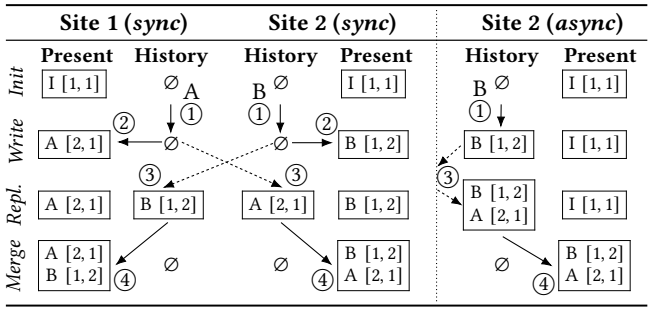


Figure 5: Evolution of the state stored in two sites for the same key. Only *data* and *lts* are pictured. Present in Site 2 *async* refers to the partial materialization in the *Present* layer.

4 A CRDV Library

This section highlights how we use the features commonly found in SQL database management systems to implement a proof-of-concept library that supports the CRDV approach.

4.1 Logical timestamp encoding

We consider three distinct strategies for encoding vector timestamps. The decision to choose one over the others will depend on the space and time overheads, measured in Section 5.2.

4.1.1 Relational-based. As a baseline, we consider a relational-based strategy that uses a separate table to store the timestamps, with each row storing the time for some site. The main advantage of this approach is that it only requires a single index to accelerate timestamp processing, independently of the number of sites. However, it requires a join to find the timestamp of a row, and the number of rows needed increases with the number of sites.

4.1.2 Vector-based. Another option is to use arrays or JSON to directly encode the timestamps. The array approach implicitly maps each site to an index, while JSON does the mapping explicitly. These encodings avoid the need to join with an auxiliary relation but require an index per site. A single index on *lts* only accelerates the "=" operator. For the ">" or "<" operators, the comparison is usually done from left to right, meaning $[0, 1]$ is considered to happen before $[1, 0]$, while in fact they are concurrent.

4.1.3 Geometric-based. The last approach uses geometry to encode timestamps. A logical timestamp of length N can be plotted as a point in N dimensions. Taking into account another point at the origin, vector timestamps form lines in one dimension (Figure 4a), rectangles in two (Figure 4b), and cuboids in three (Figure 4c). With four or more, they form hypercubes. Regardless, computing $A \leq B$ is equivalent to determining whether A is contained in B. Likewise, computing $A||B$ is equivalent to computing if A is not contained in B and B is not contained in A. Finally, determining the pointwise max of A and B is equivalent to computing the minimum bounding box of A and B. This alternative is explored, as there are several libraries to accelerate these types of geometric queries using a single index, commonly available in SQL database systems.

4.2 Replication and garbage collection

All data inserted in the *History* layer need to eventually be replicated to all sites. In our implementation in PostgreSQL, this is achieved with *publications* and *subscriptions* [21, 22], which use asynchronous logical replication. It guarantees that data from the same transaction are atomically delivered to each site. Furthermore, it ensures that missing data are replicated after a network partition.

For each site, we create a *publication* in the *History* table and *subscriptions* to the remote sites. The *publication* only replicates local Inserts, which means 1) a site can safely remove a row after it has been merged to the *Present* layer, without having Deletes propagating to remote sites, and 2) a site will not receive back its own changes from others. Therefore, this provides a straightforward solution to the garbage collection of *History*: In the *sync* mode, local *History* rows are deleted in the same trigger that performs the materialization, meaning that they are never actually stored in this layer. In systems without these properties, one could also keep track of update propagation and perform garbage collection asynchronously.

Figure 5 depicts an example of how writes are processed and replicated in the *sync* and *async* modes. In the *sync* mode, a write to *History* (①) triggers the materialization to the *Present* table (②), as well as deletion of the entry from *History*, in the same transaction. In this case, the writes at both sites ("A" and "B") replace the initial version ("I"). In contrast, a write in the *async* mode is not materialized. Independently of mode, the writes are asynchronously propagated to the *History* of the remote sites (③).

The materialization of remote data, as well as local data in the *async* mode, is achieved with a daemon implemented using the `pg_background` extension [41]. To avoid holding locks for a prolonged time, writes are split into multiple batches, which can be merged in parallel. Data originating from the same transaction are grouped in the same batch using the `xmin` system column, which identifies a transaction [25]. As with *sync* local writes, it deletes the obsolete *History* rows alongside their materialization, atomically.

In Figure 5, the remote concurrent write in the *sync* mode is placed with the existing version (④). In *async*, the old version is removed and the current ones are added. In *no-mat*, *History* would keep all versions inserted and *Present* would never be materialized.

5 Evaluation

To experimentally evaluate CRDV, we use an implementation in PostgreSQL, comparing it to other proposals that directly integrate CRDTs (Section 5.1), in a cloud environment. We begin by analyzing the various timestamp encoding strategies (Section 5.2). Then,

we demonstrate how CRDV adapts to the workload (Section 5.3). Next, we examine the overall performance of various operations (Section 5.4). Finally, we explore CRDV's scalability (Section 5.5).

5.1 Systems and environment

We implement the five structures enumerated in Section 3.2 with CRDV, relying on generic *History* and *Present* tables holding all data. Although several conflict resolution views are available, we use *LWW* in the experiments. To measure the effective overhead of CRDV, we consider SQL implementations of the same data without metadata, merge semantics, or replication, labeled *native*. For example, a *counter* is modeled as a regular integer column.

In addition to CRDV, we consider three alternatives. ElectricSQL [47] is a synchronization layer for local-first applications, using a PostgreSQL server as the source of truth and SQLite with JavaScript client-side. Writes are processed by triggers, which use *LWW* to merge conflicts, while reads retrieve the table's rows. We model *registers*, *sets*, and *maps* similarly to CRDV. To evaluate its performance, we execute direct reads/writes to the central server. Pg_crdt [79] is a PostgreSQL extension that offers CRDTs by using Automerge [9][1] – a library that implements several structures and handles conflict resolution – storing in PostgreSQL the encapsulated data. Reads/writes must be decoded/encoded by a client-side Automerge library. It supports both *LWW* (default) and *MVR* rules. Finally, Riak KV [80] is a distributed key-value database system that internally implements CRDTs. Riak KV considers *AW* for *sets* and *maps*, with *LWW* (default) and *MVR* rules for additional conflicts.

All tests run on Amazon Web Services `c7i.2xlarge` instances (8 vCPUs, 16GB Mem) with Ubuntu 22.04 LTS, unless otherwise stated. The clients run in a separate instance from the one(s) running the database server(s), deployed in the same zone. All instances use 100GB of gp3-class Elastic Block Storage (SSD, 500 MB/s, 10k IOPS).

The benchmarks are implemented in Go and use the appropriate database driver for each system. Unless otherwise stated, the result of each test is obtained by performing an average of three one-minute runs, with the first and last three seconds of each discarded. For the latency results, we also plot the 95$^{\text{th}}$ percentile (p95).

5.2 Timestamp encoding

The first tests aim to find the optimal timestamp encoding by analyzing the performance of operations against 1k items with 1k versions each (1M rows). We measure retrieving the current timestamp (Figure 6a), computing the causal present of an item (Figure 6b), and adding a new version of some item (Figure 6c). In addition, we measure the impact on storage (Figure 6d). Each strategy (Section 4.1) is evaluated by simulating 1, 2, 4, ..., and 16 sites. The benchmark uses one client and one server. To encode timestamps geometrically, we use PostgreSQL's cube extension [29].

Starting with the *current time* operation (Figure 6a), useful in both *sync* and *async*, we observe that the *array* and *json* strategies are generally faster than the *row* and *cube* alternatives. We also notice that the *row* strategy is the most affected by the increasing number of sites, as it must perform an aggregation over a progressively larger number of rows, while the others only execute index scans.

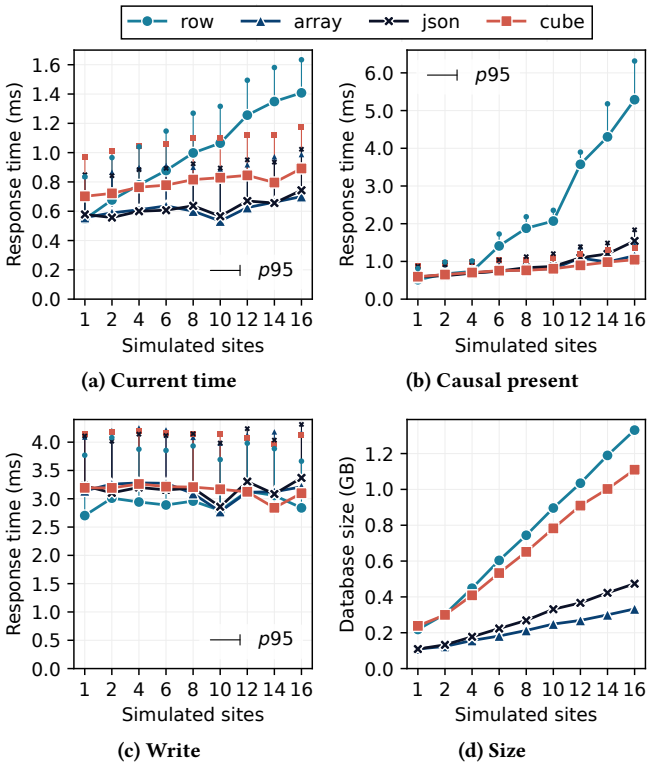


Figure 6: Comparison of different timestamp encodings.

In the *causal present* operation (Figure 6b), useful in the *async* mode,[2] the response time in all alternatives increases as new sites are added, with *row* being the most affected as the joins grow larger.

For the *write* operation (Figure 6c), useful in both modes, there is no significant difference between the alternatives. Although the *row* approach writes multiple rows, the bulk of the work in all alternatives is spent waiting for the changes to be flushed to disk.

Finally, we observe that *array* has the lowest storage overhead (Figure 6d), followed by *json*. The *row* alternative is considerably more expensive, as it stores an additional row per site. Lastly, the high storage of *cube* arises mainly from the Generalized Search Tree (GiST) index used by the cube extension, which is considerably larger than the regular B+Tree indexes used by the others.

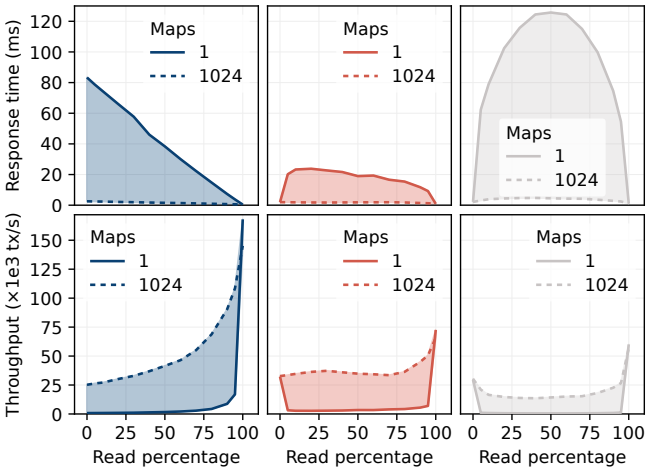
In summary, we use the *array* encoding in our implementation, as it combines low read response times with low storage requirements.

5.3 Adapting to the workload

The next tests evaluate key advantages of the CRDV approach. First, we show how different materialization strategies, made possible by the layered view architecture, impact performance in different workloads (Section 5.3.1). Second, we demonstrate the ability enabled by the CRDV approach to globally optimize an application query over the view that defines the reconciliation (Section 5.3.2). Finally, we reveal how nested structures, typical in CRDTs, are also feasible and optimized in the relational model (Section 5.3.3).

[1] Pg_crdt also supports Yjs [35] as a backend, however, it provides fewer data types.

[2] In the *sync* mode, the causal present is already materialized.



(a) *Sync* (b) *Async* (c) *No-mat*

Figure 7: Comparison of different materialization strategies in CRDV, based on the workload and number of *maps*.

5.3.1 Materialization Strategy. To evaluate the *sync* and *async* modes, we use a workload with a variable ratio of reads and writes (64 clients, one site). Reads retrieve the value of a single-key *map*, while writes update it. We also evaluate high and low contention, considering 1 and 1024 *maps*, respectively. Besides *sync* and *async*, we consider the alternative that keeps all versions and does not perform any materialization (*no-mat*) as a baseline. Figure 7 plots the response time and throughput according to the read percentage.

The high contention results (1 *map*) reveal that, with only writes, *sync* exhibits a considerably higher response time, as updates are serialized, while *async* is able to avoid conflicts. As the percentage of reads increases, *sync* response time decreases, as reads are non-blocking. In contrast, the *async* response time increases, since reads have to process numerous versions of the same item to determine the causal present. With predominantly reads, *sync* outperforms *async*, as the *async* read plan is more complex. Overall, in high-contention workloads, *sync* is superior with 80% or more reads.

With little or no contention (1024 *maps*), all response times are relatively low. However, *async* writes are still faster, since it bypasses the *merge* procedure. Overall, without contention, the *sync* approach is faster with 40% or more reads.

Finally, *no-mat* is the slowest strategy when combining reads and writes. This highlights the need to materialize the *Present* at least periodically, to keep the cost of reads manageable.

5.3.2 Plan Optimization. The following tests demonstrate how CRDV takes advantage of the query planner and the importance of the database configuration to optimize queries in CRDV. To do so, we consider an application with shopping carts – with identifier *c_id* – which store products' identifiers *p_id* and quantities *q_id*. A *LWW* view for the shopping carts is also defined, similarly to Figure 2b. Two types of queries are used, namely, "get a shopping cart's products" and "get the shopping carts containing at least one product in a defined range". These queries are evaluated using the *sync* mode, however, the same optimizations apply to the *async*.

Listing 3: Physical plans for different CRDV queries.

```
1 Subquery Scan (Time: 0.080 ms)
2  WindowAgg
3   Incremental Sort
4    Index Scan on shoppingcartpresent_pkey
5    Index Cond: c_id = x
```

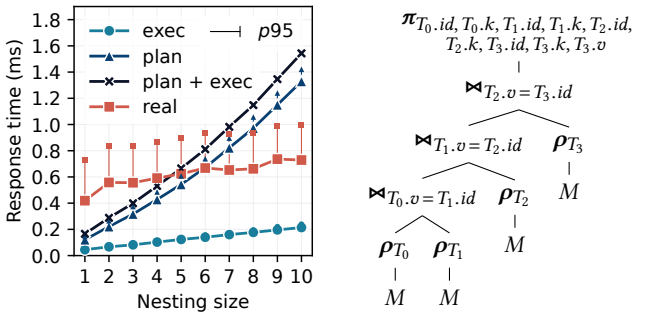
(a) `WHERE cart_id = x`

```
1 Unique (Time: 50.042 ms)
2  Subquery Scan
3   WindowAgg
4    Gather Merge
5     Sort
6      Parallel Seq Scan
7      on shoppingcartpresent
8      Filter: x ≤ p_id ≤ y
```

(b) `WHERE p_id BETWEEN x AND y`

```
1 Unique (Time: 2.266 ms)
2  Subquery Scan
3   WindowAgg
4    Sort
5     Bitmap Heap Scan
6      Bitmap Index Scan
7      on shoppingcart_p_id_idx
8      Cond: x ≤ p_id ≤ y
```

(c) b) + an index on `p_id`



(a) Response time (b) Partial logical plan ($n = 4$)

Figure 8: CRDV's read performance with different levels of nesting. *M* refers to the *MapAwLww* view.

The physical plan of the first query, shown in Listing 3a, shows the planner pushing the selection directly to the *Present* table, leveraging the existing index (lines 4, 5) and avoiding materializing the entire view. Meanwhile, the second query's plan (Listing 3b) shows the planner scanning the entire table (lines 6-8), requiring a relatively long execution time (50ms). To improve this query, we index `p_id` in the *Present* table. Now, the resulting physical plan (Listing 3c) reveals that the planner commences execution by first selecting only the rows whose products are in the required range (lines 6-8), leading to a 22× improvement over the initial version.

This shows that we can apply to conflict-free data the same optimization techniques as used in regular relational workloads. With solutions relying on opaque structures, such as Pg_crdt or Riak KV, the same optimizations are not easily achieved (Section 5.4).

5.3.3 Nested Structures. We now examine the performance of reading nested structures, commonly found in practical applications of CRDTs [38]. Unlike solutions that encapsulate nested structures, CRDV needs to perform a join to retrieve the next level, which can be costly if these joins are not optimized. We perform reads with one client and one site to 100k total *maps* with 1, 2, …, and 10 levels of nesting. Each level in a *map* links to another *map*. The read query also materializes the complete structure as JSON. Figure 8a plots the response time for each level, while Figure 8b shows a partial

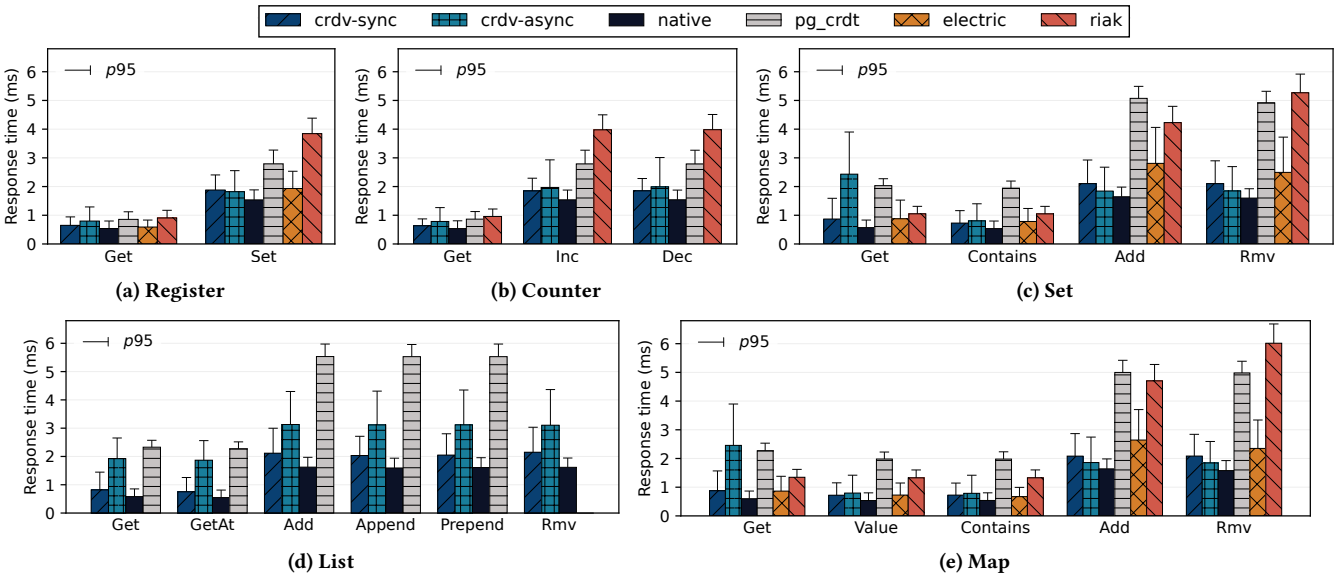


Figure 9: Performance comparison between different operations in different structures, using different solutions. *Get* - read the structure; *GetAt* - list's element at some index; *Contains* - check if an element/key exists in a *set*/*map*; *Value* - value of some key in a *map*; *Set* - update a *register*; *Inc/Dec* - increment/decrement a counter; *Add* - add a new entry to a *set*/*list*/*map* or update a *map's* value; *Prepend/Append* - insert an entry at the start/end of a list; *Rmv* - remove an entry from a *set*/*list*/*map*. The missing structures/operations are not supported by their respective solutions.

logical plan. These results measure the optimization time (`plan`), the execution time (`exec`), and the time seen by the client (`real`).

The results show that the execution time (`exec`) increases only slightly when more nesting levels are added. This is due to the optimization done to the plan, which incrementally joins each level with the next, as shown in Figure 8b, taking advantage of the existing index. Meanwhile, the planning time (`plan`) increases significantly as more levels are added, being at most 6× more expensive than the execution time. However, the actual response time seen by the client (`real`) is lower than the combined plan and execution times. This is attributed to the query caching employed by PostgreSQL, which significantly minimizes the overall response time.

Once again, these results demonstrate how the performance of CRDV is considerably improved by the existing query optimizer.

5.4 Operations

The next tests measure the performance of common operations on CRDTs, using one client and one site. We evaluate both the *sync* and *async* modes of CRDV, as well as alternatives. To generate a large dataset, we populate each data type with 100k items, with each *set*, *list*, and *map* having an initial size of 100 (~30M rows). Figure 9 plots the average response time for each operation.

Comparing first the *sync* and *async* modes of CRDV, we once again conclude that reads are faster in the *sync* mode, especially when retrieving entire collections, i.e., *get* of *set*/*list*/*map*; and, conversely, writes are faster in the *async* mode. The main exceptions are the *counter* and *list* writes, which also require a read to compute the current value and determine the correct list index, respectively. The reads in *sync* mode are also similar in response time to *native* and ElectricSQL, as they only perform an index scan. Likewise, the *async* writes are also similar to *native*'s. ElectricSQL's more expensive writes stem from the triggers that perform the materialization, which synchronously update multiple metadata tables.

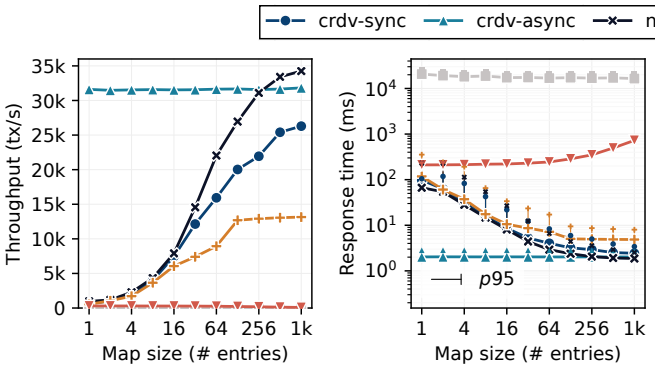
Meanwhile, both Pg_crdt and Riak KV have higher read response times. Looking at multi-item structures, such as *sets*, it is also visible that retrieving a single element is just as expensive as retrieving the entire structure (e.g., comparing *Get* with *Contains*). This is a drawback of the opaque design, which makes it impossible to query the internal structure directly, forcing the client to retrieve and decode the entire structure. The writes in these systems are also generally more expensive, as the entire object must be rewritten. In Pg_crdt, the structure must also be retrieved first to be updated. Likewise, *rmv* operations to *sets* and *maps* in Riak KV also require an initial read to determine the causal context [82, 83].

Overall, CRDV's *sync* strategy has a response time overhead of around 30% compared to the *native* SQL implementation. Meanwhile, ElectricSQL, PG_crdt, and Riak KV are around 5%, 50%, and 40% slower than CRDV, respectively.

5.5 Scalability

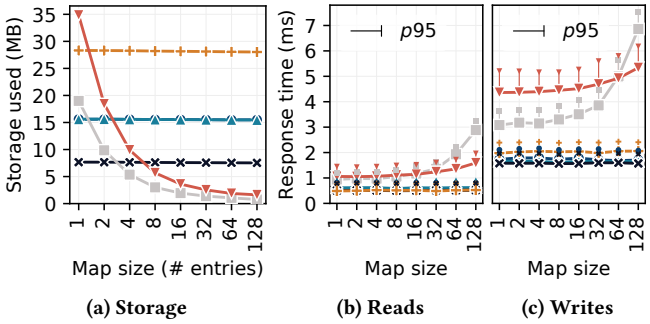
The next series of tests study the scalability of CRDV, comparing it with other systems. We explore the impact of hotspots on performance (Section 5.5.1), the amount of storage required (Section 5.5.2), the impact of each operation on the network (Section 5.5.3), the replication performance, including with network partitions (Section 5.5.4), and the performance of horizontal scaling (Section 5.5.5).

5.5.1 Concurrency. We start by analyzing the impact of hotspots on scalability by executing updates to one *map* with a variable number of entries – from 1 to 1024 – using 64 clients and one site. Figures 10a and 10b plot throughput and latency, respectively.



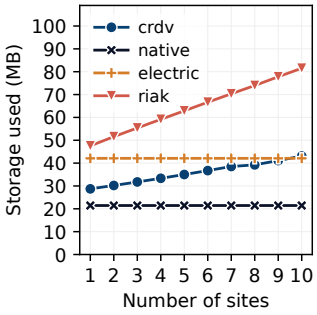
(a) Throughput (b) Response time

Figure 10: Write performance of different solutions in a variable contention workload, using 64 clients.



(a) Storage (b) Reads (c) Writes

Figure 11: Storage usage and latency of different solutions with 100k key-value pairs, based on the total number of *maps*. $x=1 \rightarrow$ 100k *maps* of size 1, $x=2 \rightarrow$ 50k *maps* of size 2, and so on.



(a) Total storage per site

	Attribute	Bytes/row
Data	*key* + *data* (w/index)	213
Metadata	*op*	1
	site	4
	pts	33
	lts (w/index)	21 + 16 × site
Real overhead vs. *native*		34% (1 site) 41% (2 sites)

(b) CRDV storage details

Figure 12: Storage usage based on the number of sites.

Excluding CRDV's *async*, the throughput of all purely relational systems is low when the number of *map* entries is small, as writes are sequential when targeting the same row. However, this quickly changes with the increasing number of entries, as writes can now access different rows, thus improving concurrency. CRDV's *async* mode is, in contrast, not affected by the number of entries, as its lazy materialization approach avoids serialization. ElectricSQL's lower scalability is due to the relatively high number of operations executed. For example, in addition to the UPDATE in the main table, there are four additional UPDATES to the metadata tables.

In contrast, the performance of Pg_crdt and Riak KV does not improve with the increasing number of entries, with both lines located near 0. Once more, the opaque implementation means that updates to the same structure are serialized even when targeting different entries. Thus, for these systems, it is recommended to split hotspot structures into multiple distinct objects.

5.5.2 Storage. We now evaluate how the different systems scale storage-wise. To do this, we perform two types of tests. The first evaluates how storage in each system behaves based on the number of elements in each structure, while the second compares the storage overhead based on the number of sites.

Starting with the first test, we consider 100k total key-value pairs and a variable number of *maps*. For example, when *map size* = 1, there are 100k *maps* with 1 entry; when *map size* = 2, there are 50k *maps* with 2 entries each; and so on. Figure 11 plots the storage required based on the *map* size (Figure 11a), with accompanying read (Figure 11b) and write (Figure 11c) response times, using one client and one site. The reads retrieve an entry from a *map*, while the writes update an entry (one client and one site).

Starting with CRDV and the other purely-relational systems, we conclude that they are not impacted by how the dataset is organized. Independently of the number of *maps*, they will always store 100k total rows. Here, CRDV requires on average 2× more space than *native*, while ElectricSQL requires 1.8× more than CRDV. Although ElectricSQL materializes the data like the *native* implementation, the bulk of the storage derives from its metadata tables.

For Pg_crdt and Riak KV, their storage requirements are high when there are many different structures with a few items each, which is caused by the size of the metadata stored by each structure. Conversely, with only a few structures, the storage requirements are considerably lower compared to the relational systems. This is the main benefit of the opaque structure implementation, as there is no metadata stored for each entry. On the other hand, this incurs higher read (Figure 11b) and write (Figure 11c) latencies, as the complete object needs to be decoded and encoded, respectively.

Moving to the second test, we consider 100k single-item structures. To measure the expected overhead in a real-life scenario, we base the data on the average row size in the TPC-C benchmark [86] (around 210 bytes, with indexes). We test CRDV with 1 to 10 sites, comparing it to the baseline with no metadata (*native*). Additionally, we also compare against Riak KV (1 to 10 sites) and ElectricSQL (single site). Figure 12a shows the total storage based on the number of sites, while Figure 12b details the storage usage in CRDV.

The results in Figure 12a demonstrate that the storage of CRDV and Riak KV increases with the number of sites, a consequence of vector clocks. Riak KV increases at a higher rate, as it also attaches the site identifiers.[3] Pg_crdt, which follows a strategy similar to that of Riak KV, is not evaluated since it stores all operations, making it difficult to determine the overhead. Meanwhile, a system such

[3]It is worth pointing out that this allows Riak KV to store a site's timestamp only when it writes to it. In these tests, each site has written once to each object.

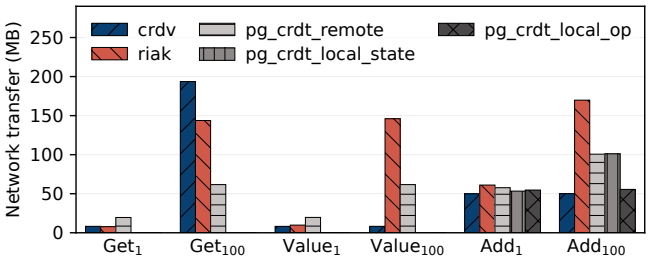


Figure 13: Network overhead of different distributed solutions, based on 100k operations performed to *maps*. X_1 and X_{100} represent operations done to *maps* with 1 and 100 entries, respectively.

as ElectricSQL, which operates solely under *LWW*, does not use vector clocks and would therefore not be affected by the number of sites. Overall, with one site, CRDV has an overhead of 34% over the baseline (Figure 12b). However, most is fixed and unaffected by the number of sites. For example, the array to store the vector clock uses 21 bytes without any elements. For each new site, we have an additional 16 bytes per row – to store the logical timestamp and the respective index entry – which means the increase is only 7%.

5.5.3 Network. The following tests measure the impact of CRDV on the network. That is, the data transferred from one site to the client and from one site to another. We again rely on a dataset with 100k *maps*, using one client and two sites. For each operation type, we execute one run until 100k total operations have been completed. Each operation is redirected to the same site, where we monitor the network usage. To compare with CRDV (*sync*), we deploy Riak KV with two replicas [84], and Pg_crdt, deployed with its central server (*pg_crdt_remote*). We also consider a local-first deployment with Pg_crdt as the source of truth, using a SQLite database in the client for reads and writes, and asynchronous replication to and from the central server. The asynchronous replication from the central server to the client can either send the full object (*pg_crdt_local_state*) or only the *changes* (*pg_crdt_local_op*), simulating state and operation-based replication, respectively.[4] Figure 13 shows the total data transferred for each operation, considering *maps* of size 1 and 100.

Starting with the *get* operation, which retrieves a *map*, we see that CRDV transfers an amount of data proportional to the size, as expected. Meanwhile, both Pg_crdt *remote* and Riak KV transfer a lower amount than CRDV, which is on par with the storage results of Figure 11a. As for the local-first solutions, the data transferred is zero, as the client does not communicate with the central server.

Moving on to the *value* operation, which retrieves a single key-value pair, it shows that the data transfer in CRDV is the same independently of the total number of entries. In contrast, Pg_crdt *remote* and Riak KV must retrieve and decode the entire object, which explains the increased network usage with larger objects.

Finally, for the *add* operation, which updates one entry, we examine that the bandwidth used by Pg_crdt *remote*, Pg_crdt *local_state*, and Riak KV increase with the size of the *map*, due to replicating the entire object. Both CRDV and the operation-based version of

[4]In all Pg_crdt versions, when updating an object, the client only sends the *changes* to the remote server instead of the entire object, to reduce the amount of data transferred.

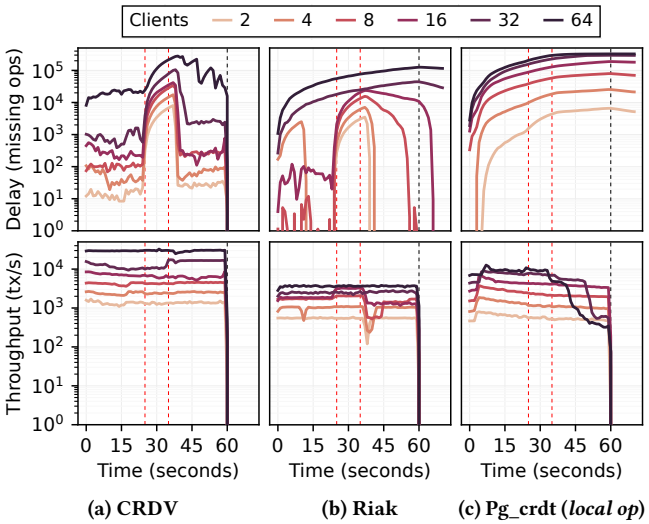


Figure 14: Delay and throughput over time of different distributed solutions. The network is partitioned between t=25 and t=35.

Pg_crdt transfer the same amount of data independently of the structure size. This shows that while CRDV is state-based, its network overhead resembles operation-based solutions since it operates row by row. However, unlike operation-based systems such as Pg_crdt's *local_op*, there is no need for idempotency or causal delivery [71].

5.5.4 Freshness. The next tests assess CRDV's ability to promptly process remote data while engaging with local workloads. To measure the delay between one site and the others, we assign each client a fixed number of *counters* to update. With different clients accessing different sites, each site will effectively manage different *counters*. The workload consists of increments to some *counter*, with periodic reads to log the current values. The delay at some site is later determined by analyzing the logs and comparing the values read. We use three sites and a variable number of clients, running one 60-second run with a network partition between the sites from second 25 to 35, to measure how the system copes with long backlogs. CRDV (*sync*) uses a single merge process with a batch size of 10k. To compare, we rely on Riak KV (three sites) and the local-first Pg_crdt deployment defined in Section 5.5.3 (operation-based; each client has its own SQLite database)[5]. Figure 14 plots the delay at one site and the combined throughput over time.

CRDV's results show a consistent baseline delay in all loads, caused by the latency between the sites and the merge delay of the background worker. This means that the sites are not expected to drift apart under normal circumstances. Even after the network partition, the delay is reduced relatively quickly to normal levels. The key attribute enabling these results is the fact that CRDV merges data in batches, i.e., multiple rows are merged in a single transaction, and hence the amortized disk sync cost is virtually zero. To cope with higher loads, the number of merge processes can be increased.

As for Riak KV, it can consistently handle up to 16 clients. After that, the sites become increasingly out-of-sync. With Pg_crdt,

[5]In Pg_crdt, the network is partitioned between the clients and the central server.

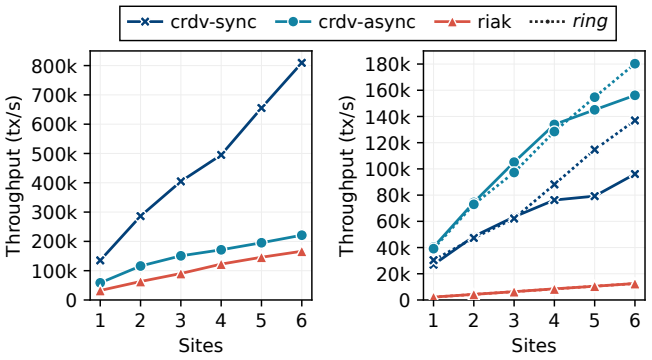


Figure 15: Performance comparison based on the cluster size.

the clients become out-of-sync when running with just 2, as local operations are generated faster than the remote server can handle. Additionally, due to long replication queues, Pg_crdt runs out of memory with 32 and 64 clients and starts to rely on swap.

In summary, CRDV consistently keeps the delay low while maintaining a higher throughput than the competing alternatives.

5.5.5 Multiple sites. The final scalability tests evaluate CRDV based on the number of sites. We separately execute reads (Figure 15a) and writes (Figure 15b) to 1M *registers*, using 1 to 6 sites. To guarantee a sufficient load, we run 1k clients in a `c7i.4xlarge` instance (16 vCPUs). To compare with CRDV, we also deploy Riak KV. In addition to the regular fully connected topology, we test the write workload using a circular topology (*ring*).

Starting with the read results (Figure 15a), both CRDV *sync* and Riak KV scale linearly based on the number of sites, although Riak KV has a significantly lower baseline. For CRDV *async*, its performance increase is slightly affected by the size of the timestamps.

Regarding the write results (Figure 15b), both CRDV strategies start with linear scalability but slow down after 4 sites. With a fully connected topology, the replication overhead is proportional to the number of sites since each sends data to and receives from every other. With the circular topology (*ring*), both strategies are now still efficient. However, the replication delay will increase and the availability will depend on the ability to reconfigure the ring. Riak KV shows a linear write scalability but a low baseline. We noticed that the CPU usage was low, while the number of disk operations was high, equal to the benchmark's throughput. This means that each write is separately synced to disk, which limits performance. Meanwhile, PostgreSQL uses *group commit* to flush multiple transactions in a single sync [19], amortizing the disk cost.

In summary, these results show that CRDV scales reads optimally, while for writes they highlight the need to choose an appropriate topology to reduce the impact of replication on performance.

6 Related Work

Handling Conflicts. Avoiding and handling conflicts has long been acknowledged as key to efficient replication in database systems. Namely, Gray et al. in a seminal paper on database replication [17] propose precisely a two-tier system that defers updates but still relies on a central system. Asynchronous replication in SQL database systems allows the definition of reconciliation rules for conflicting updates but does not avoid false conflicts as it does not recognize causality. The *last-writer-wins* rule [85] was commonly adopted by eventually consistent systems to merge conflicting writes and is still the main way to do it in datastores such as DynamoDB [10] and Cassandra [43]. Moreover, Dynamo [10] proposed a *multi-value register* for each data item, delegating reconciliation to the application. However, they do not use a query optimizer, as they are simple key-value stores.

Conflict-free Replicated Data Types. Much of the research on conflict-free replication that goes beyond the basic collection structures has been framed in the context of CRDTs, namely, to model counters while enforcing lower limits [5], to represent lists with a variety of algorithms [58, 74, 87, 88], and to implement generic JSON documents with nesting support [38]. As we have shown, the flexibility of the relational model allows us to support these structures and take advantage of the same algorithms in CRDV.

The commutative properties of CRDTs have been particularly useful in local-first systems [39], such as collaborative text editors [36, 59, 78]. To this end, several libraries providing CRDTs have been developed, such as Automerge [9], Yjs [35], and Diamond Types [15]. Local-first database systems offering CRDTs have also been created, such as OrbitDB [8] and RxDB [51]. OrbitDB stores operations in a Merkle-Tree-based log, sorted using *LWW*. To read an object, it iterates over the log until it finds the respective entry, unlike CRDV. The RxDB CRDT plugin stores the array of operations with the respective document. Like OrbitDB, all operations are considered while conflicting writes are deterministically handled by selecting the one with the highest creator identifier. CRDTs have also found their place in server-side distributed database systems, such as Riak KV [81], Redis [73], and AntidoteDB [2], providing a variety of data types. However, unlike CRDV, they do not offer customizable conflict resolution rules, and the opaque values of the data model do not allow for fine-grained projections and selections.

Other research on CRDTs has pointed to their integration directly with the query planner [42]. M. Kleppmann further conjectures that CRDTs can be specified as a query on a view that includes all write operations and proposes modeling them in Datalog [37]. CRDV tackles similar issues but targets practical scenarios, namely, using common database features to encode conflict-free data and implement read and write paths, introducing and demonstrating the relevance of customizable materialization to handle diverse workloads, and separating replication details from conflict resolution to allow diverse data types and criteria to be easily implemented.

CRDTs in Relational Systems. There has been a push to bring these data types to the relational model. Pg_crdt [79] stores binary blobs – representing CRDTs – in rows and relies on the Automerge or Yjs libraries. The binary representation means increased latency due to the constant serialization/deserialization and makes it hard to use strategies such as indexing. Another set of solutions model the tables themselves as CRDTs, such as ElectricSQL [47], Conflict-free Replicated Relations [90] (CRR), CR-SQLite [89], and Dart sql_crdt [6]. All of these rely on a similar approach, i.e., tagging each row with a timestamp and using *LWW* to handle conflicts, eagerly materializing the relations. CRR also supports *counters*, but its

Table 2: Summary of CRDV and related solutions.

Solution	Data Model	Interface	Implementation	Sync.	Conflict Resolution	Optimization
CRDV	Relational	SQL	Views+Triggers	State	Custom (SQL)	Yes
Automerge [9]	Document	Custom API	Library (Multi-lang)	Operation[a]	LWW, MVR	No
Yjs [35]	Document	Custom API	Library (Multi-lang)	Both[b]	Highest Id	No
D. Types [15]	Text	Custom API	Library (Rust, JS)	Operation	-	No
OrbitDB [8]	Key-value	Custom API	Library (JS)	Operation[c]	LWW	No
RxDB [51]	Document	MQL[d]	Library (JS)	Operation	Highest Id	Simple[e]
Riak KV [81]	Key-value	Custom API	Internal (Erlang)	State	AW, EW, LWW, MVR	No
Redis [73]	Key-value	Custom API	Internal (C)	Operation	AW, LWW	No
AntidoteDB [2]	Key-value	Custom API	Internal (Erlang)	Operation	[A|R]W, [E|D]W, LWW, MVR	No
AntidoteSQL [49]	Relational	AQL	Library (Erlang)	Operation	[A|R]W, EW, LWW	Simple[e]
Pg_crdt[f] [79]	Relational	Custom API	PG Ext., Lib. (Multi-lang)	Operation	LWW, MVR	No
ElectricSQL [47]	Relational	SQL	Lib. (JS), Triggers+Views	State	LWW	Partial[g]
CRR [90]	Relational	SQL	Library (Erlang)	State	LWW	Partial[g]
CR-SQLite [89]	Relational	SQL	SQLite Extension	State	AW, LWW	Partial[g]
Dart sql_crdt [6]	Relational	SQL	Library (Dart)	State	LWW	Partial[g]
Lasp [50]	Key-value	Custom API	Library (Erlang)	State	Custom (Erlang)[h]	No

[a] As documents store all changes, it also supports replicating the entire state. [b] Operation-based for inserts, state-based for deletes [34]. [c] While the synchronization is operation-based, each operation contains the entire state. [d] MongoDB Query Language. [e] Support secondary indexes but are otherwise limited when compared to major relational DBMSs. [f] Using Automerge as the backend. [g] Unlike CRDV, conflict resolution is not considered by the optimizer. [h] Unlike CRDV, rules must defined at build time.

implementation has not been made available. Thus, and since it has a server-side implementation in PostgreSQL, we chose ElectricSQL to evaluate experimentally this class of solutions. Just as CRDV, ElectricSQL also relies on triggers to handle writes. Unlike CRDV's multi-layered view architecture, these solutions only consider a limited set of structures and conflict resolution rules, do not cater for configurable materialization, indexing, and optimization, and rely on external components to, for example, handle replication.

High-Level Semantic Customization. Similarly to CRDV, there are solutions based on manipulating CRDTs through a high-level interface. AntidoteSQL [49] provides an extended subset of SQL (AQL) interface to AntidoteDB that allows creating tables with specific merge rules, namely *AW* and *RW* for rows, handling column conflicts with *EW* for flags and *LWW* for the rest. Additionally, it supports *counters*. Unlike CRDV, it is not possible to specify custom resolution rules. Furthermore, its optimization capabilities are limited to secondary indexes, and since it does not support a standard interface, its usage is challenging outside Erlang. Lasp [50] is an applicational-level functional programming model that allows describing the semantics of CRDT using a common interface. Unlike CRDV, it is not integrated into the database, making it more difficult to reuse across applications. More generally, other solutions enable high-level customization of isolation and consistency. For example, some systems support changing the isolation [32], labeling operations and/or specifying invariants over the data to select the appropriate consistency level [3, 4, 40, 45, 46, 48, 77], or defining transactional guarantees through the query language [13].

Table 2 provides a high-level comparison between CRDV and alternatives. In summary, it is the only solution that provides custom runtime resolution rules – using a high-level declarative interface – and allows them to be optimized together with user workloads. CRDV's novelty also extends to supporting custom data materialization strategies on conflict-free replicated data.

7 Conclusions and Future Work

Although there has been substantial work proposing new database management systems supporting CRDTs [2, 8, 51, 80], we show that this can be achieved in mainstream SQL database systems through the use of widely available features, such as views to support user-defined resolution rules at runtime, and triggers/rules and asynchronous replication to exchange and merge data.

Interestingly, our proposal allows for more flexibility in the definition of merge strategies and results in better performance, not in spite of the old tried and tested SQL system but precisely by exploiting its advantages: Abstraction in declarative programming with SQL and views, global query optimization, and the ability to fine-tune access paths with indexing and materialization. A key detail was expressing causality [44] in the relational model by defining and benchmarking three different encodings that are amenable to indexing and query optimization.

This work opens up interesting research directions. First, we envision that CRDV can be extended to support cross-site ACID guarantees, which would provide a framework to explore the impact of combining strong and eventual transactions over the same data. Second, as data processing in CRDV is done fully in the relational model, it better supports the analysis of arbitrary queries on CRDTs as proposed by Laddad et al. [42]. Finally, it would be interesting to explore how the existing database features can be improved for large-scale heterogeneous deployments, where data might be partitioned and/or some sites might be deployed at the edge.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their thoughtful comments, which helped to improve our work. This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. DOI 10.54499/LA/P/0063/2020.

References

[1] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 405–414. doi:10.1109/ICDCS.2016.98

[2] AntidoteDB. 2024. AntidoteDB Documentation - Datatypes in Antidote. https://antidotedb.gitbook.io/documentation/architecture/datatypes.

[3] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination avoidance in database systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 185–196. doi:10.14778/2735508.2735509

[4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[5] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending eventually consistent cloud databases for enforcing numeric invariants. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 31–36.

[6] Daniel Cachapa et al. 2024. Dart sql_crdt: Dart implementation of Conflict-free Replicated Data Types (CRDTs) using SQL databases. https://github.com/cachapa/sql_crdt.

[7] E. F. Codd. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.* 4, 4 (dec 1979), 397–434. doi:10.1145/320107.320109

[8] OrbitDB Community. 2024. OrbitDB: Peer-to-Peer Databases for the Decentralized Web. https://github.com/orbitdb/orbitdb.

[9] Automerge contributors. 2024. Automerge: A JSON-like data structure (a CRDT) that can be modified concurrently by different users, and merged again automatically. https://automerge.org/.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. doi:10.1145/1323293.1294281

[11] Nuno Faria and José Pereira. 2021. Totally-Ordered Prefix Parallel Snapshot Isolation. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data* (Online, United Kingdom) *(PaPoC '21)*. Association for Computing Machinery, New York, NY, USA, Article 6, 7 pages. doi:10.1145/3447865.3457966

[12] Nuno Faria and José Pereira. 2023. MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting. *Proc. ACM Manag. Data* 1, 1, Article 43 (may 2023), 27 pages. doi:10.1145/3588723

[13] Nuno Faria, José Pereira, Ana Nunes Alonso, Ricardo Vilaça, Yunus Koning, and Niels Nes. 2023. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2274–2288. doi:10.14778/3598581.3598598

[14] C. J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

[15] Seph Gentle et al. 2024. Diamond Types: The world's fastest CRDT. https://github.com/josephg/diamond-types.

[16] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (June 2002), 51–59. doi:10.1145/564585.564601

[17] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data - SIGMOD '96* (Montreal, Quebec, Canada). ACM Press, New York, New York, USA. doi:10.1145/233269.233330

[18] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 13.3. Explicit Locking. https://www.postgresql.org/docs/16/explicit-locking.html.

[19] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 20.5. Write Ahead Log. https://www.postgresql.org/docs/16/runtime-config-wal.html.

[20] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 31. Logical Replication. https://www.postgresql.org/docs/16/logical-replication.html.

[21] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 31.1. Publication. https://www.postgresql.org/docs/16/logical-replication-publication.html.

[22] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 31.2. Subscription. https://www.postgresql.org/docs/16/logical-replication-subscription.html.

[23] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 31.6 Logical Replication Restrictions. https://www.postgresql.org/docs/16/logical-replication-restrictions.html.

[24] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 40. Event Triggers. https://www.postgresql.org/docs/16/event-triggers.html.

[25] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 5.5. System Columns - xmin. https://www.postgresql.org/docs/16/ddl-system-columns.html#DDL-SYSTEM-COLUMNS-XMIN.

[26] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 7.2.1.5. LATERAL Subqueries. https://www.postgresql.org/docs/16/queries-table-expressions.html#QUERIES-LATERAL.

[27] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 8.14. JSON Types. https://www.postgresql.org/docs/16/datatype-json.html.

[28] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – 8.15. Arrays. https://www.postgresql.org/docs/16/arrays.html.

[29] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – F.11. cube — a multi-dimensional cube data type. https://www.postgresql.org/docs/16/cube.html.

[30] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – SQL Commands: CREATE TRIGGER. https://www.postgresql.org/docs/16/sql-createtrigger.html.

[31] The PostgreSQL Global Development Group. 2024. PostgreSQL Documentation – SQL Commands: CREATE VIEW. https://www.postgresql.org/docs/16/sql-createview.html.

[32] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)* 15, 4 (1983), 287–317.

[33] Joseph M Hellerstein and Peter Alvaro. 2020. Keeping CALM: when distributed consistency is easy. *Commun. ACM* 63, 9 (Aug. 2020), 72–81. doi:10.1145/3369736

[34] Kevin Jahns et al. 2024. Yjs Internals. https://github.com/yjs/yjs/blob/main/INTERNALS.md.

[35] Kevin Jahns et al. 2024. Yjs: Shared data types for building collaborative software. https://github.com/yjs/yjs.

[36] Project Jupyter. 2024. JupyterLab Documentation. https://jupyterlab.readthedocs.io/en/latest/index.html.

[37] Martin Kleppmann. 2018. Data structures as queries: Expressing CRDTs using Datalog. https://martin.kleppmann.com/2018/02/26/dagstuhl-data-consistency.html.

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

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

[40] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment* 2, 1 (2009), 253–264.

[41] Vibhor Kumar. 2024. Postgres Background Worker Exntesion. https://github.com/vibhorkum/pg_background.

[42] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M Hellerstein. 2022. Keep CALM and CRDT on. *Proceedings VLDB Endowment* 16, 4 (Dec. 2022), 856–863. doi:10.14778/3574245.3574268

[43] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[44] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. Association for Computing Machinery, New York, NY, USA, 179–196. doi:10.1145/3335772.3335934

[45] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 281–292.

[46] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[47] Electric DB Limited. 2024. ElectricSQL - Sync for Modern apps. https://electric-sql.com.

[48] Jed Liu, Tom Magrino, Owen Arden, Michael D George, and Andrew C Myers. 2014. Warranties for faster strong consistency. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 503–517.

[49] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Ségio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno Preguiça. 2019. Antidote SQL: Relaxed When Possible, Strict When Necessary. (Feb. 2019). arXiv:1902.03576 [cs.DB] http://arxiv.org/abs/1902.03576

[50] Christopher Meiklejohn and Peter Van Roy. 2015. Lasp: A language for distributed, coordination-free programming. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. 184–195.

[51] Daniel Meyer et al. 2024. RxDB: A fast, local first, reactive Database for JavaScript Applications. https://github.com/pubkey/rxdb.

[52] Microsoft. 2024. SQL Server Documentation – CREATE TRIGGER (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-ver16.

[53] Microsoft. 2024. SQL Server Documentation – DDL Triggers. https://learn.microsoft.com/en-us/sql/relational-databases/triggers/ddl-

triggers?view=sql-server-ver16.
[54] Microsoft. 2024. SQL Server Documentation – JSON data in SQL Server. https://learn.microsoft.com/en-us/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver16.
[55] Microsoft. 2024. SQL Server Documentation – Merge Replication. https://learn.microsoft.com/en-us/sql/relational-databases/replication/merge/merge-replication?view=sql-server-ver16.
[56] Microsoft. 2024. SQL Server Documentation – Views. https://learn.microsoft.com/en-us/sql/relational-databases/views/views?view=sql-server-ver16.
[57] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 511–524.
[58] 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.
[59] Matthieu Nicolas, Victorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat, and François Charoy. 2017. MUTE: A peer-to-peer web-based real-time collaborative editor. In *ECSCW 2017-15th European Conference on Computer-Supported Cooperative Work*, Vol. 1. EUSSET, 1–4.
[60] Oracle. 2024. MySQL Reference Manual – 13.5 The JSON Data Type. https://dev.mysql.com/doc/refman/8.4/en/json.html.
[61] Oracle. 2024. MySQL Reference Manual – 19 Replication. https://dev.mysql.com/doc/refman/9.0/en/replication.html.
[62] Oracle. 2024. MySQL Reference Manual – 27.3 Using Triggers. https://dev.mysql.com/doc/refman/8.4/en/triggers.html.
[63] Oracle. 2024. MySQL Reference Manual – 27.6 Using Views. https://dev.mysql.com/doc/refman/9.0/en/views.html.
[64] Oracle. 2024. Oracle 8 Concepts – 31. Database Replication. https://docs.oracle.com/cd/A58617_01/server.804/a58227/ch_repli.htm.
[65] Oracle. 2024. Oracle SQL Language Reference – 15. CREATE TRIGGER. https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/CREATE-TRIGGER.html.
[66] Oracle. 2024. Oracle SQL Language Reference – 15. CREATE TRIGGER Statement - ddl_event. https://docs.oracle.com/en/database/oracle/oracle-database/23/lnpls/CREATE-TRIGGER-statement.html#GUID-AF9E33F1-64D1-4382-A6A4-EC33C36F237B__CIHBDEFD.
[67] Oracle. 2024. Oracle SQL Language Reference – 15. CREATE VIEW. https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/CREATE-VIEW.html.
[68] Oracle. 2024. Oracle SQL Language Reference – 2. Data Types - JSON Data Type. https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/Data-Types.html#GUID-E441F541-BA31-4E8C-B7B4-D2FB8C42D0DF.
[69] Oracle. 2024. Oracle SQL Language Reference – 2. Data Types - Varrays. https://docs.oracle.com/en/database/oracle/oracle-database/23/sqlrf/Data-Types.html#GUID-EAA3885B-06AA-4F0D-85E7-C43352E5E2AC.
[70] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.
[71] Nuno Preguiça. 2018. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254* (2018).
[72] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, asynchronous schema change in F1. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1045–1056.
[73] Redis. 2024. Redis Enterprise - Active-Active geo-distribution. https://redis.io/active-active/.
[74] 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.
[75] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://inria.hal.science/inria-00555588
[76] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 386–400.
[77] Krishnamoorthy C Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. *ACM SIGPLAN Notices* 50, 6 (2015), 413–424.
[78] Nathan Sobo. 2022. How CRDTs make multiplayer text editing part of Zed's DNA. https://zed.dev/blog/crdts.
[79] Supabase. 2024. Pg_crdt - POC CRDT support in Postgres. https://github.com/supabase/pg_crdt.
[80] Basho Technologies. 2024. Riak KV. https://riak.com/products/riak-kv/.
[81] Basho Technologies. 2024. Riak KV Documentation - Data Types. https://docs.riak.com/riak/kv/latest/developing/data-types/index.html.
[82] Basho Technologies. 2024. Riak KV Documentation - Data Types: Maps. https://docs.riak.com/riak/kv/2.2.3/developing/data-types/maps.1.html.
[83] Basho Technologies. 2024. Riak KV Documentation - Data Types: Sets. https://docs.riak.com/riak/kv/2.2.3/developing/data-types/sets/index.html.
[84] Basho Technologies. 2024. Riak KV Documentation - V3 Multi-Datacenter Replication Reference: Architecture. https://docs.riak.com/riak/kv/2.2.3/using/reference/v3-multi-datacenter/architecture/index.html.
[85] Robert H Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)* 4, 2 (1979), 180–209.
[86] Transaction Processing Performance Council (TPC). 2010. *TPC Benchmark™ C Standard Specification Revision 5.11.* Technical Report. Transaction Processing Performance Council. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf
[87] Matthew Weidner, Joseph Gentle, and Martin Kleppmann. 2023. The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing. *arXiv preprint arXiv:2305.00583* (2023).
[88] Stephane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE transactions on parallel and distributed systems* 21, 8 (2010), 1162–1174.
[89] Matt Wonlaw et al. 2024. Cr-SQLite: Convergent, Replicated SQLite. Multi-writer and CRDT support for SQLite. https://github.com/vlcn-io/cr-sqlite.
[90] Weihai Yu and Claudia-Lavinia Ignat. 2020. Conflict-free replicated relations for multi-synchronous database management at edge. In *2020 IEEE International Conference on Smart Data Services (SMDS)*. IEEE, 113–121.