

Keep CALM and CRDT On

Shadaj Laddad*

University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*

University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano

University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung

University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks

University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein

University of California, Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Despite decades of research and practical experience, developers have few tools for programming reliable distributed applications without resorting to expensive coordination techniques. Conflict-free replicated datatypes (CRDTs) are a promising line of work that enable coordination-free replication and offer certain eventual consistency guarantees in a relatively simple object-oriented API. Yet CRDT guarantees extend only to data updates; observations of CRDT state are unconstrained and unsafe. We propose an agenda that embraces the simplicity of CRDTs, but provides richer, more uniform guarantees. We extend CRDTs with a query model that reasons about which queries are safe without coordination by applying monotonicity results from the CALM Theorem, and lay out a larger agenda for developing CRDT data stores that let developers safely and efficiently interact with replicated application state.

1 INTRODUCTION

Consistency is a central theme of distributed computing research, with major implications for practitioners. Modern cloud-hosted applications are frequently distributed to optimize for latency and availability. When application state is replicated across the globe, developers often face stark choices regarding replica consistency. Strong consistency can be enforced in a general-purpose way at the storage or memory layer via classical distributed coordination (consensus, transactions, etc.), but this is often unattractive for latency and availability reasons. Alternatively, application developers can build on “weakly” consistent storage models that do not use coordination; in this case developers must reason about consistency at the application level.

The last decade has seen a surge of research interest in reasoning about application consistency, featuring everything from complex formal invariants [54] to multi-tiered consistency annotations [19, 40] to explicit happens-before annotations on operations [12]. In recent years, one approach has risen above the noise among practitioners: Conflict-Free Replicated Data Types [52]. CRDTs are provided as an API by a few commercial platforms (e.g., Enterprise Redis, Akka, Basho Riak [7, 27, 48]), and have been documented in use by a number of products and services including PayPal, League of Legends, and Soundcloud [6, 18, 38]. There is also a growing set of open source CRDT packages that have thousands of stars on

GitHub [25, 26, 43], and blog posts explaining CRDTs to developers in pragmatic, informal terms [13, 49, 57].

The attractiveness of CRDTs lies in their combination of (1) an easy-to-explain API, and (2) the promise of formal safety guarantees. Designing a CRDT centers around providing a function to *merge* any two replicas, with the requirement that this single function is associative, commutative and idempotent (ACI), and defining atomic *operations* that clients can use to update a replica. From the user’s perspective, the CRDT’s object-oriented API often mimics a familiar collection; many of the CRDTs in the literature are simple adaptations of well-known data types like Sets and Counters.

The formal safety properties of CRDTs, as originally phrased by Shapiro et al, leverage “a well defined interface ... [with] mathematically sound rules to guarantee state convergence” [52]. This guarantee is achieved via the ACI properties of the merge function. Classic anomalies in eventually consistent systems are caused by reordered, duplicated, or late-arriving updates—none of which can affect the result of an idempotent, commutative, and associative function execution.

But this strong convergence guarantee addresses only state updates and offers no APIs (or guarantees!) for *visibility* into the state of a CRDT. Although useful queries are often included in the presentation of CRDT designs, these have no impact on the correctness of the CRDT and are no safer to use than arbitrary queries executed directly on the underlying state. In one of the precursor papers to CRDTs that also proposes ACI merge functions, Helland and Campbell go as far as noting ironically that READs are “annoying” and may not commute with other actions [15].

EXAMPLE 1 (THE POTATO AND THE FERRARI, A.K.A. EARLY READ). *A canonical CRDT is the Two-Phase Set (2P-Set) [51], which is a pair of sets (A, R) that track items to be added (A) and removed (R). The merge function for two 2P-Sets is defined simply as the pairwise union, $(A_1 \cup A_2, R_1 \cup R_2)$ and is patently ACI. This scheme was used in the well-known Amazon Dynamo shopping cart example [11].*

Implicit in this design is a query $Q = A - R$ returning the intended contents of the set. Consider a scenario where a shopper adds a potato and a Ferrari to their cart, then removes the Ferrari, and “checks out” by computing the query Q . In one or more replicas of the 2P-Set, the checkout request could arrive before the removal of the sports car. This truly expensive consistency bug arises when the query “reads” the state of the 2P-Set “too early”, before all the removals have eventually arrived. And there is no way to know that all the removals have indeed arrived without the coordination that CRDTs supposedly do not need.

In practice, the soundness of state convergence in CRDTs does not translate to predictable guarantees for computations that examine them. One might say that CRDTs provide *Schrödinger consistency*

* equal contribution



guarantees: they are guaranteed to be consistent only if they are not viewed.

The weak consistency of CRDT queries is not a secret in the research literature [63, 64], and is mentioned in the initial papers [51, 52]. At the same time, bloggers and other developer-facing venues have latched onto the formal language of the initial papers (“principled” [52], “mathematically sound” [52], “theoretically sound” [51]), and sometimes without caveats. For example, one online article argues that CRDTs “allow multiple replicas in different regions to mathematically resolve to the same state without coordination ... multiple active copies present accurate views of the shared datasets at low latencies” [20]. This dangerous misread of CRDT guarantees suggests that more work is needed to ensure developers use CRDTs safely.

We believe that the gaps in CRDT guarantees can be addressed on two fronts: (a) defining more precisely what developers must reason about when using CRDTs in their applications and (b) building data systems for CRDTs that automatically manage replication and query execution to deliver stronger consistency guarantees. The unconstrained nature of queries in CRDTs raises an intriguing question: can we develop a more formal query model that makes it possible to precisely define when execution on a single replica yields consistent results?

In this paper, we explore how the CALM (Consistency As Logical Monotonicity) theorem [3], originally formulated as a definition for consistency in distributed logic programs, can be used as the basis of a query model for CRDTs that delineates queries that can be executed locally from those that require coordination among a quorum. Because monotonicity can be identified as a static property, this view of queries paves the path for a CRDT data system that provides efficient *and safe* execution of queries. Guided by this vision, we map out a research path that weaves together query optimization, storage abstractions, provenance, and more to bring the coordination-free benefits of CRDTs to developers while preserving the consistency guarantees they expect.

2 AN OVERVIEW OF CRDTS

Most discussions of CRDTs begin by introducing two functionally equivalent representations: *state-based* CRDTs (a.k.a. *CvRDTs*) and *op-based* CRDTs (a.k.a. *CmRDTs*). Essentially, state-based CRDTs gossip data, while op-based CRDTs gossip logical log records.

2.1 State-Based CRDTs

We begin by reviewing the definition of state-based CRDTs. CvRDTs encapsulate the current *state* of the replica; let the type of *state* be called *T*. The API for state-based CRDTs contains three classes of methods, all of which run locally on a single replica’s state:

Merge: merge is a single, required method that takes a value *v* of type *T* as input. It combines *state* with *v* to generate a value *state'* of type *T*, and updates itself so that *state* = *state'*. *Constraint*: the merge function must be ACI.

Operations: these are methods that clients use to modify *state*. *Constraint*: operations must be monotonic with respect to the type *T*.

Queries: these are methods that do not modify *state*, but return a result that may be dependent on *state*.

```
merge(adds, removes) {
  state.adds = union(state.adds, adds);
  state.removes = union(state.removes, removes);
}
operation add(i) { state.adds = union(state.adds, Set(i)); }
operation remove(i) { state.removes = union(state.removes, Set(i)); }
query contents() { return diff(state.adds, state.removes); }
```

Figure 1: Pseudocode for a Two-Phase Set CRDT.

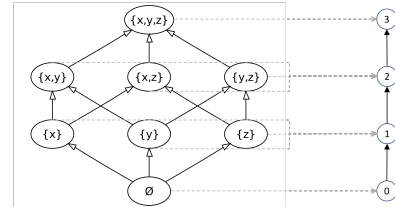


Figure 2: Hasse diagrams for G-Set and a cardinality counter, and a monotone function between them (dashed lines).

An example of a 2P-Set CvRDT is shown in Figure 1. In CvRDTs, nodes gossip their replicas to each other and apply merge upon receipt of gossip. If we focus only on *state* and merge, a CvRDT is simply a new name for a classical mathematical construct: the *join semi-lattice*. A join semi-lattice is defined in precisely the same way: it is a pair $S = (D, \sqcup)$ where *D* is a domain (i.e. type) and \sqcup is an operation (called “join” or “least upper bound”) that is ACI.

A well-known property of a join semi-lattice *S* is that it is isomorphic to a partial order \leq_S on *D* such that given two elements $s, t \in D$, $s \leq_S t$ iff $s \sqcup t = t$. (We will drop the subscript on \leq when it is clear from context.) The familiar “Hasse diagrams” for semi-lattices capture this by laying out the elements of *D* on a y-axis corresponding to the \leq ordering (Figure 2). Note that this order is partial: two elements $s, t \in D$ may be incomparable: $s \not\leq t \wedge t \not\leq s$.

Operations are monotonic with respect to the partial order \leq : if an operation replaces *state* with *state'*, we require that *state* \leq *state'*. A good way to enforce this is to forbid operations from modifying *state* directly, and instead require them to invoke merge to perform the state update—the ACI properties of merge then ensure monotonic updates. Viewed through that lens, CvRDTs only update their state through the ACI merge function, and are precisely join semi-lattices [30].

Note that although queries are included in the API surface of CvRDTs, they are unconstrained and may perform arbitrary computations on the underlying state. Therefore, the choice of queries included with a CRDT design have no effect on the *safety* of observations through them—their consistency guarantees are no stronger than a single query that just emits the internal state of the CRDT.

2.2 Op-Based CRDTs and Compression

The idea of op-based CRDTs (CmRDTs) is to gossip logs of operations rather than state. Each given replica *r* applies its operations sequentially, so upon gossip we require that another replica *r'* applies the log records from *r* in an order that produces the same final state as *r*. In typical deployments of op-based CRDTs, this restriction is imposed by requiring the network between replicas to

guarantee causal delivery. As a result, the logs accumulated at every replica follow the standard *happens-before* partial order defined by Lamport [32].

But as noted above, a partial order like happens-before is isomorphic to a join semi-lattice! That means that we can capture the CmRDT log as a CvRDT. One simple way to do this is to hold the DAG corresponding to the partial order: this is simply a “grow-only” set—whose only operation is add—which holds edges of a DAG. Each directed edge connects two operations, and indicates that the operations happened in the order of the edge. (Another solution would be to use CvRDTs for vector clocks.) A natural query over such a CvRDT is to “play the log”: i.e. produce a topological sort of the log and apply the operations in the resulting order.

CmRDTs can therefore be considered a gossip compression technique that maps instances of one CvRDT (say a 2P-Set as in Figure 1) to another CvRDT (a partially ordered log of add and remove operations). Hence CmRDTs are arguably a specific form of a CvRDT for partially-ordered logs, with various tricks applied to suit the nature of the operations being logged. In the remainder of our discussion, we focus on the clean lattice-based CvRDT API, and use the CRDT acronym to mean CvRDTs.

2.3 A Snapshot of CRDTs in Practice

This model of CRDTs has gained traction in the industry across a wide range of applications ranging from high-scale backend logic to client-side collaborative state. Before we dive into our vision for extensions to CRDTs that make them safer to use, let us briefly explore the ways they are already being applied.

CRDT designs in use today largely fall into two buckets: the core CRDTs from the early literature designed to mimic classic data structures [51] and more advanced CRDTs focused on replicating documents for collaborative editing [60, 61]. Across a variety of languages, developers have created several libraries [25, 42] that provide high-quality implementations of the core CRDTs. These CRDTs have also become adopted as building blocks that can be used by distributed systems developers through systems like Akka [48], Dynamo [11], and Redis [7], which all provide CRDTs as built-in data structures. There are also well documented examples of industry players building systems on top of CRDTs, such as PayPal [38] and League of Legends [18]. We even see examples of CRDTs used inside of databases such as in FlightTracker [53] at Facebook, which uses CRDTs to provide stronger consistency guarantees in the TAO data store.

A significant portion of recent CRDT research has focused on *collaborative editors*, which have latency and fault-tolerance challenges that CRDTs are well-suited to address. The research in this space is primarily interested in representing the variety of ways that users interact with text documents, such as text insertion, cut/paste to different locations, and formatting layers. A lot of recent creativity in the CRDT space has gone into this domain, resulting in designs such as Peritext [35].

3 TOWARD A QUERY MODEL FOR CRDTs

In Section 1 we argued that CRDTs have gained interest for their combination of safety, efficiency, and simplicity. In that spirit, our desiderata for a good CRDT query model are:

Safety: Queries should be sequentially consistent, regardless of the replica at which they are evaluated.

Efficiency: Queries should be evaluated locally without coordination whenever possible.

Simplicity: The query model should be easy for developers to reason about.

3.1 Example Queries

Let’s look at some examples of queries that can and cannot be executed without coordination while satisfying sequential consistency.

EXAMPLE 2 (A BOOLEAN THRESHOLD QUERY OVER A GROW-ONLY SET). *One of the simplest possible CRDTs is the Grow-Only Set (G-Set) [51]. It is a set S with an operation that can add elements to S and a merge function that is the set union, $S_1 \cup S_2$.*

Say we want to determine whether the number of gift card purchases that are over \$100 in a set of transactions has exceeded 50 items (similar to the threshold functions in LVars [29]):

```
query suspicious_activity() {
  if (cardinality([
    txn for txn in state
    if txn.type == "GIFTCARD" and txn.amount > 100
  ]) > 50) {
    return true else ABORT;
  }
}
```

Note that the `suspicious_activity` query returns either true or aborts (signifying “unknown”). Perhaps surprisingly, executing this query on a local replica will always produce a sequentially consistent result, even without coordination! This is because each replica of a CRDT effectively represents an *under-approximation* of some true global state; that is, each individual replica has seen some *subset* of the updates which have entered the system at any given time. Thus, the true “global” state of the system can be derived from any individual replica’s local state by adding in some number of missing update operations. This query’s true result *cannot be changed* by any subsequent updates; thus, for the purposes of *this query*, our replica’s local state reflects the same information as a true “global” state would contain, and can serialize after that global state. If the query aborts, we learn nothing; aborted queries are not considered when determining sequential consistency.

Let’s return to our Potato/Ferrari example from the introduction. In this example, the query we wanted to ask, $Q = A - R$, would not yield a consistent result when executed locally. We again are guaranteed that *eventually* the local state will converge to the global state and give us the correct answer to the query, by the eventually consistent properties of CRDT operations. However, *unless we can ensure that our local state is equal to the global state via coordination*, we cannot determine that the result of our local query matches the result of the global query. We could be missing the effect of an operation locally (e.g., `remove(‘Ferrari’)`) and we would output an incorrect result.

Consider a third query, again over the 2P-Set. This time we want to rate-limit a shopping cart user by detecting whether the number of actions they have taken exceeds 100. Our query is $Q = |A| + |R| > 100$. This query can be computed locally without coordination by the same reasoning as `suspicious_activity`: since A and R

individually are guaranteed to increase in size, we know that the $global(|A| + |R|) \geq local(|A| + |R|)$.

3.2 What’s Going on Here? ... Monotonicity!

What is going on here? Some queries are consistent without coordination, but others require a global view of the system to be correct? The difference between these queries is that the threshold queries are **monotone** queries with respect to the CRDT state and its partial order, whereas the shopping cart checkout query is **non-monotone**. This distinction may be familiar to the reader from the CALM Theorem [3, 16], which proved that programs are convergent without coordination iff they are monotone.

The CALM Theorem is framed in terms of monotonicity over logic on relations—but it applies equally well to the CRDT domain! Per Section 2, we know that both the merge and update methods of a CRDT monotonically increase the value of the CRDT’s state with respect to its partial order. We can define a monotone query as any whose output is monotone with respect to ordering of the CRDT. That is, given a join semi-lattice $S = (D, \sqcup)$ as the state of the CRDT, a query Q is monotone if $\forall i, j \in D : i \leq j, Q(i) \implies Q(j)$. By the CALM Theorem, monotone queries over CRDTs are exactly the queries that only need a local view of the system to be correct!

Monotone queries meet all the criteria we outlined for a good query model. They are able to achieve safety and efficiency for queries over CRDTs. The CALM Theorem tells us that not only do they meet these criteria (if), but they are the only queries that meet this criteria (only if). Further, monotone queries offer composition through monotone functions. As previously observed (Bloom^L [8], Lasp [39], Datafun [4]) this compositionality allows construction of complex systems out of CRDT primitives, and is highly amenable to programming language tools. The space of monotone queries is quite large; for example, four of the five operators of relational algebra are monotone: selection, projection, union, and intersection. Only set difference is non-monotone. Observe that a pipeline composing monotone functions will always give a monotone function end-to-end, but if the pipeline contains any non-monotone function then the end-to-end-computation will be non-monotone.

Perhaps the most important property, since it focuses on the adoption barriers for CRDTs, is the simplicity of our query model. We believe that query monotonicity should be understandable to anyone who understands basic CRDTs. The state in standard CRDT examples is either sets or counters, both of which have simple, intuitive definitions of monotonicity. Moreover, the definition of CRDTs requires developers to understand monotonicity with respect to state updates, so it should be reasonable for developers to extend this reasoning to include queries as well. Because monotonicity can be syntactically identified in many existing query languages, including SQL, we are optimistic that developer tools can help guide the creation of monotonic queries.

3.3 Beyond Monotonicity

Monotonic queries are the natural query model for CRDTs; their resilience to update re-ordering mirrors the convergent nature of updates within a CRDT. We acknowledge, however, that there are large classes of queries performed on CRDTs which are *not* monotonic. A simple example of a non-monotonic query is set

difference ($Q = A - R$), seen in our shopping cart. In non-monotonic queries, missing global information can make results appear to go *backwards* over time, making it impossible to safely make decisions based on the results of these queries.

So what are developers to do when they need one of these non-monotone queries? The simple and safe solution is to coordinate! Queries executed against global state, after all, will always be correct. The choice of coordination falls into a classic spectrum for distributed databases outlined by Bernstein and Goodman [5]: write-one read-all, write-majority read-majority, or write-all read-one. Each of these strategies, however, is still improved by the use of CRDTs: with CRDTs, only non-monotone queries need to be ordered, with respect to *sets* of updates. As update operations commute, there is no need to coordinate in order to sequence update operations with each other.

Developers building on CRDTs retain the option to perform a local, but stale query on a single replica. The resulting application-level considerations align well with those established by high-scale systems developed in industry, such as Google’s Zanzibar authorization system [45], which offers APIs for retrieving safe, up-to-date results or recent, but potentially stale ones. For applications that can tolerate out-of-date results, with a staleness distribution determined by the gossip protocol, local non-monotone queries offer a fast path that can reduce the overall latency perceived by the user.

4 ENABLING FAST AND SAFE CRDT SYSTEMS

Equipped with our distinction of monotone queries, we believe that developers will be able to apply CRDTs in new ways by developing complex applications on top of replicated state. By reducing query correctness to a simple property, monotonicity, developers can reason about the correctness of their data architectures and know the pitfalls to avoid when creating CRDT-backed systems. As more developers write software that depends on CRDTs in complex ways, it is critical that the research community explores methods that help developers utilize CRDTs robustly and efficiently. To this end, we propose a shift in perspective from an object-oriented view of CRDTs to a “database” view of them: breaking CRDTs up into a query model and a data store that separates their logical and physical representations. This shift in perspective brings us to many impactful research problems in data management for CRDTs.

4.1 A Query Language For CRDTs

In Section 3, we outlined a formal query model for CRDTs that uses monotonicity as a key property to determine how the query should be executed to guarantee consistent results. Our next challenge is to identify how this model can be mapped to a practical language that developers can use to declare the observations they want to make on CRDTs.

What we need in a query language is a set of rich expressions that can manipulate the lattice structures used inside CRDTs, and a syntax that makes it easy for developers (and computers) to identify when a query is monotone and can be efficiently executed. One promising choice is to develop a dialect of the query language most developers are already familiar with: SQL! Recent theoretical work [22] demonstrated an extension of relational queries to

operate over semi-rings. We are currently working on a similar formalization for extending relational algebra to semi-lattices.

There are two major benefits of using the SQL language for queries over CRDTs. First, it is clear syntactically in SQL queries whether the query is monotone or not, which will help developers design efficient programs and reason about them. Second, we already know how to build optimizers for SQL that take advantage of monotonicity! Streaming joins and barrier stages are examples of an optimizer leveraging its knowledge of monotonicity and non-monotonicity respectively in the dataflow graph.

Using relational-style languages to query CRDTs also fits well into recent research exploring alternate models of CRDTs. In particular, there has been a recent push to define CRDTs as Datalog queries over sets of operations being gossiped across nodes [23]. In such a model, issuing queries in a relational-style language naturally fits into the execution model, and opens up the opportunity for further end-to-end optimizations such as using incremental view maintenance to identify efficient ways to propagate the effects of operations to queries.

By defining a query language that allows developers to safely view the state of a CRDT, we arrive at an interaction model that is distinct from the object-oriented, in-memory CRDTs used today. Our model of CRDTs includes application-specific operations, which are fundamental to proving convergence of the CRDT, but the lack of predefined queries deviates significantly from the classic object-oriented model. Our CRDTs can be viewed as effectively having one open-ended endpoint that all queries go through.

4.2 A Data Management System for CRDTs

Separating the capabilities of CRDTs from the object-oriented interface they are typically wrapped in raises an intriguing opportunity: operations and our query language can become an interface between application logic and an independent CRDT data store that manages all aspects of the CRDT life-cycle. We believe that this approach can both increase the ease of use of CRDTs, by shifting the responsibility of reasoning about consistency to the store, and improve the efficiency of applications built on CRDTs, since data stores can make advanced optimization decisions based on the dynamic workload. Compared to existing data stores that support CRDTs but do not provide query APIs [7, 9], our monotonic query model enables rich observations of CRDTs with strong consistency guarantees.

In our vision, a CRDT data store sits in the application stack at a similar level as a traditional relation database or key-value store. Application services can interact with the CRDT store over the network, using a protocol that can be easily implemented by several languages so that heterogeneous application implementations can interact on shared CRDT state. By deploying the CRDT store as a separate networked service, our approach also enables serverless applications to build on top of replicated state [55], since the replicas will be maintained separately from the ephemeral execution nodes.

Shifting CRDTs from being baked into application logic to being managed by an external service does raise a challenge in the extensibility with respect to the available data types. While today's CRDTs are defined as regular data types in the application language and are immediately usable, bringing custom types to

a CRDT data store requires a pluggable interface to provide the data store with an implementation of the CRDT. We believe that this can be achieved with a lightweight extension API, similar to Postgres extensions [56], that uses a foreign-function interface to inject custom CRDTs. Because we only rely on the core CRDT properties of monotonic state and convergent operations, the system can immediately make the CRDT available as long as the developer certifies it satisfies these properties.

4.3 Optimization Opportunities in CRDT Stores

The abstraction of placing CRDTs in a store separate from the application reveals a range of opportunities for research that optimizes how the CRDTs are stored and queried. Having applied our first major database trick, a query model, the next natural one to apply is the separation of logical and physical data representations. By separating these for the CRDT data model, we open up many research directions for optimizing the physical layer of CRDTs in the DBMS. An increasingly popular research topic for CRDTs is how to minimize their memory footprint [47]. Recent work applied columnar compression techniques from databases to collaborative editor CRDTs [24]. The more general question of how to automatically find compressed representations of CRDTs is open.

The other approach to minimizing the memory footprint that is ripe for automated management is garbage collection: when can you delete or compact older operations in the CRDT? In today's applications, where CRDTs are freely passed around as regular objects, it can be difficult to trace down all replicas. But when all CRDT state is managed by the data store, garbage collection can be safely employed because it is clear where replicas of each CRDT lie. Garbage collection requires advanced program analyses to determine when components of CRDT state become inaccessible. We are especially excited about the opportunity to apply program synthesis and automated verification research to discover strategies for garbage collection.

Finally, there are several enticing research topics focused on how the effects of operations are propagated between replicas. The classic algorithm for gossip with state-based CRDTs calls for the *entire* state of each replica to be sent over the network. For large CRDT instances, this places a significant networking burden that can increase staleness. But with bookkeeping that captures which versions of the state have already been gossiped, we can instead propagate smaller deltas that capture the effects of new operations. There are several intriguing research directions in this space, such as identifying compact data structures for the bookkeeping, the selection of what delta information to gossip, how to batch those deltas, and what gossip architecture and frequency to use.

4.4 Tradeoffs for Non-Monotone Queries

Although monotone queries make the most of CRDTs by entirely eliminating coordination, not all business logic can be expressed strictly in terms of such queries. Consistently executing non-monotone queries requires the CRDT data store to introduce coordination among replicas. However, not all hope for low-latency queries is lost!

As discussed in Section 3.3, datastores would only need to coordinate in anticipation of a non-monotone query. With accurate metrics and a model to predict both the frequency of these queries and the data such queries will touch, future datastores can perform much of the work of reaching consistency in advance of this non-monotone query, shifting into and out of a synchronize-on-update model accordingly (as previously observed in [33, 34]). We further note, however, that the *metrics themselves* are likely to be monotone—and thus the work of determining when to coordinate is itself amenable to our monotone query model, and can therefore avoid synchronization.

4.4.1 Weakening consistency for non-monotonicity. Some applications may prefer to avoid coordination entirely, despite the fact that coordination in the presence of non-monotonic queries is essential for maintaining consistency. These applications have essentially decided that *weak consistency* is tolerable for their purposes. Much previous work exists in the space of weakly-consistent datastores, both in terms of how to best supply usable weak consistency to the user [1, 9, 31, 37, 62] and how to help the user decide *when* weakening consistency may be appropriate [14, 41, 54].

Our framework of monotonic queries over CRDTs can enhance these existing approaches. Most obviously, it can recognize patterns of weakly-consistent reads which form monotonic queries, and can thus allow such patterns without sacrificing consistency. More subtly, building CRDTs into a database enables the use of lineage techniques that have been well-studied in existing database systems. In particular, a CRDT-and-monotonicity-aware dataflow technique would be capable of determining the potential *downstream effects* of a weakly-consistent query, providing database users and administrators alike with valuable insights into the consistency of not just their data, but the observations *derived* from that data.

This information has particular use when applied to the *apologies* strategy first proposed by Helland and Campbell in [15]. With apologies, potentially-inconsistent observations are accompanied by *compensating actions*, which are intended to clean up any negative effects of weak consistency. By leveraging lineage tracing, a CRDT-enabled database could automatically determine when such apologies are necessary, prompting the application accordingly. Indeed, this strategy is already present in PL-focused solutions such as [41].

5 RELATED WORK

CRDTs are well-studied in the research community; seminal citations mentioned earlier include [51, 52], as well as many papers on collaborative-text CRDTs [35, 49, 60, 61]. We choose to focus on CRDTs due to their rising popularity; this framing drives our choice to explore queries over CRDTs, and in turn excludes lines of work which focus on safely directly observing weakly-consistent shared memory such as [12, 54, 58]. In particular, solutions which rely on causal [37], linearizable [17], or other explicit consistency models stronger than eventual consistency (such as [9, 19, 28, 34, 40, 46, 59, 64]), were set aside in our discussion.

These papers attempt to reduce the number of anomalies that may emerge from weakly-consistent applications by eliminating certain reorderings on some operations; in contrast, we seek a simple unifying principle by which a developer can consider a

CRDT observation to be reliable, *without* reasoning about varying classes of inconsistency under which it may be read. Some work, such as [28, 33], uses CRDT-like reasoning to automate the choice of consistency model; we believe that these papers fit with our goal of discovering classes of monotonic observations safe to perform with weak consistency, and we seek to extend them here.

While we focus on how developers *use* CRDTs, there is a wide range of research on how these data types are *designed*. Recent work has explored how CRDT designs can be generated using program synthesis techniques [30], which significantly reduces the burden of designing new types for custom application logic. In addition, a few replicated data types have been proposed as alternatives to CRDTs, such as ECROs [10] and MRDTs [21], with the focus of making it easier for developers to express the semantics of the merge logic. Like traditional CRDTs, however, these are also focused on convergent updates to the data type, and leave queries unconstrained. Other work, such as [64], introduce total order to certain CRDT operations in order to allow consistent observations; in contrast, we attempt to identify a class of *monotonic* observations which do not require total order. Similarly, other work solves the consistent read problem in ways reminiscent of escrow transactions [44] or single-master replication [36, 50].

Perhaps the most related work comes from the programming languages and databases space, with languages such as Gallifrey, LVars, Lasp, Datafun, and Bloom^L all providing capabilities for constraining monotonic logic [4, 8, 29, 39, 41]. In Section 4, we discuss a SQL-like query language with a database optimizer and intelligent storage layer. Almost all of the research directions we outline in that setting also apply to the compiler, runtime, and storage layer for monotonicity-aware DSLs like these.

We believe that lighter-weight solutions than rewriting applications in a new DSL or query language will also play an important role in safe adoption of CRDTs. One promising such solution is type annotations in existing languages along the lines of Blazes [2]. In a DSL designed for CRDT queries, similar type systems could be used to enforce monotonicity while supporting complex query logic. Other angles from the programming languages and software engineering communities include formal verification and fuzzing, which can assist developers designing CRDTs by automatically checking the core correctness properties.

6 CONCLUSION

CRDTs are simple, and beginning to see adoption among developers—an important signal for database researchers. CRDTs on their own lack power. However, the research literature abounds with results—such as the CALM theorem—which when combined with CRDTs open new frontiers to researchers and developers alike. The next generation of challenges in this space will arise at the seams between foundational research and practical concerns. These challenges will require research spanning data management, distributed systems, query models, and programming languages.

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)*. IEEE, 405–414.
- [2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and David Maier. 2014. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*. 52–63.
- [3] Tom J Ameloot, Frank Neven, and Jan Van den Bussche. 2013. Relational transducers for declarative networking. *Journal of the ACM (JACM)* (2013), 1–38.
- [4] Michael Arntzenius and Neelakantan R Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 214–227.
- [5] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* (1981), 185–221.
- [6] Peter Bourgon. 2014. Roshi: a CRDT system for timestamped events. (May 2014). <https://developers.soundcloud.com/blog/roshi-a-crdt-system-for-timestamped-events>
- [7] The Redis Community. 2022. Redis. <https://redis.io>
- [8] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*.
- [9] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDis: A Branch-and-Merge Approach To Weak Consistency. In *Proceedings of the 2016 International Conference on Management of Data*. 1615–1628.
- [10] Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: Building Global Scale Systems from Sequential Code. *Proc. ACM Program. Lang.* (2021).
- [11] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voss, and Werner Vogels. 2007. Dynamo: Amazon’s highly available key-value store. *ACM SIGOPS operating systems review* (2007), 205–220.
- [12] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. 2019. Verifying C11 programs operationally. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 355–365.
- [13] Sph Gentle. 2022. 5000x faster CRDTs: An Adventure in Optimization. <https://josephg.com/blog/crdts-go-brrrr/>, retrieved June 1, 2022.
- [14] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. ‘Cause I’m strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 371–384.
- [15] Pat Helland and David Campbell. 2009. Building on quicksand. *arXiv preprint arXiv:0909.1788* (2009).
- [16] Joseph M Hellerstein and Peter Alvaro. 2020. Keeping CALM: when distributed consistency is easy. *Commun. ACM* (2020), 72–81.
- [17] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1990), 463–492.
- [18] Todd Hoff. 2014. *How League Of Legends Scaled Chat To 70 Million Players - It Takes Lots Of Minions*. <http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html>
- [19] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 279–293.
- [20] Leena Joshi. 2022. How to simplify distributed app development with CRDTs. <https://techbeacon.com/app-dev-testing/how-simplify-distributed-app-development-crdts>, retrieved May 31, 2022.
- [21] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* (2019).
- [22] Mahmoud Abo Khamis, Hung Q. Ngo, Reinhard Pichler, Dan Suciu, and Yisun Remy Wang. 2022. Convergence of Datalog over (Pre-)Semirings. In *PODS*.
- [23] Martin Kleppmann. 2018. Data structures as queries: Expressing CRDTs using Datalog. (2018). <https://martin.kleppmann.com/2018/02/26/dagstuhl-data-consistency.html>
- [24] Martin Kleppmann. 2019. Experiment: columnar data encoding for Automerge. (2019). <https://github.com/automerge/automerge-perf/blob/master/columnar/README.md>
- [25] Martin Kleppmann. 2022. Automerge. <https://github.com/automerge/automerge>
- [26] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Transactions on Parallel and Distributed Systems* (2017), 2733–2746.
- [27] Rusty Klopheus. 2010. Riak Core: Building Distributed Applications without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming*.
- [28] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 113–126.
- [29] Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*. 71–84.
- [30] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. <https://doi.org/10.48550/ARXIV.2205.12425>
- [31] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* (2010), 35–40.
- [32] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* (1978), 558–565.
- [33] Cheng Li, Joao 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.
- [34] 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)*.
- [35] Geoffrey Litt, Slim Lim, Martin Kleppmann, and Peter van Hardenberg. 2021. Peritext: A CRDT for Rich-Text Collaboration. <https://www.inkandswitch.com/peritext>
- [36] 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.
- [37] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.
- [38] Dmitry Martynov. 2018. CRDTs in Production. <https://www.infoq.com/presentations/crdt-production>
- [39] 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.
- [40] Mae Milano and Andrew C. Myers. 2018. MixT: A Language for Mixing Consistency in Geodistributed Transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 226–241.
- [41] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. 11:1–11:19.
- [42] Tyler Neely and David Rusu. 2022. crdts: family of thoroughly tested hybrid crdt’s. <https://github.com/rust-crdt/rust-crdt>
- [43] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2015. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Proceedings of the 15th International Conference on Engineering the Web in the Big Data Era - Volume 9114*. 675–678.
- [44] Patrick E O’Neil. 1986. The escrow transactional method. *ACM Transactions on Database Systems (TODS)* (1986), 405–430.
- [45] Ruoming Pang, Ramón Cáceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christopher D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google’s Consistent, Global Authorization System. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) (USENIX ATC ’19). USENIX Association, USA, 33–46.
- [46] Krithi Ramamritham and Calton Pu. 1995. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering* (1995), 997–1007.
- [47] Arik Rinberg, Tomer Solomon, Roei Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma. 2022. DSON: JSON CRDT using delta-mutations for document stores. *Proceedings of the VLDB Endowment* (2022), 1053–1065.
- [48] Raymond Roostenburg, Rob Williams, and Robertus Bakker. 2016. *Akka in action*.
- [49] Colin Rofls. 2022. CRDT - The Xi Text Engine. <https://xi-editor.io/docs/crdt-details.html>, retrieved June 1, 2022.
- [50] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1311–1326.
- [51] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types*. Ph.D. Dissertation. Inria—Centre Paris-Rocquencourt; INRIA.
- [52] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. 386–400.
- [53] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. {FlightTracker}: Consistency across {Read-Optimized} Online Stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 407–423.
- [54] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *Proceedings of the 36th*

- ACM SIGPLAN Conference on Programming Language Design and Implementation. 413–424.
- [55] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452. <https://doi.org/10.14778/3407790.3407836>
- [56] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) (SIGMOD '86). Association for Computing Machinery, New York, NY, USA, 340–355. <https://doi.org/10.1145/16894.16888>
- [57] Bartosz Sypytkowski. 2022. An Introduction to State-based CRDTs. <https://bartoszsypytkowski.com/the-state-of-a-state-based-crds/>, retrieved June 1, 2022.
- [58] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 309–324.
- [59] Chao Wang, Constantin Enea, Suha Orhun Mutluergil, and Gustavo Petri. 2019. Replication-Aware Linearizability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 980–993.
- [60] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2007. Wooki: A P2P Wiki-Based Collaborative Writing Tool. In *8th International Conference on Web Information Systems Engineering*. 503–512.
- [61] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*. 404–412.
- [62] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412.
- [63] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*. 75–87.
- [64] Xin Zhao and Philipp Haller. 2018. Observable Atomic Consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 23–32.