Wrapping Rings in Lattices: An Algebraic Symbiosis of Incremental View Maintenance and Eventual Consistency

Conor Power
UC Berkeley
conorpower@cs.berkeley.edu

Saikrishna Achalla
UC Berkeley
saikrishna.achalla@berkeley.edu

Ryan Cottone
UC Berkeley
rcottone@berkeley.edu

Nathaniel Macasaet
UC Berkeley
nmacasaet1003@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

Abstract

We reconcile the use of semi-lattices in CRDTs and the use of groups and rings in incremental view maintenance to construct systems with strong eventual consistency, incremental computation, and database query optimization.

Keywords: distributed systems, eventual consistency, incremental view maintenance, query optimization, algebraic systems

ACM Reference Format:
Conor Power, Saikrishna Achalla, Ryan Cottone, Nathaniel Macasaet, and Joseph M. Hellerstein. 2024. Wrapping Rings in Lattices: An Algebraic Symbiosis of Incremental View Maintenance and Eventual Consistency. In *The 11th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3642976.3653031

1 Introduction

Algebraic models have been growing in popularity recently in both distributed systems and databases. In distributed systems, the semi-lattice model popularized as conflict-free replicated data types (CRDTs) offers an algebraic perspective on strong eventual consistency. In the database community, a generalization of relational algebra to semi-rings was promoted in the study of database provenance and a similar view in terms of full rings has been used to study database incremental view maintenance (IVM) [8, 18]. Both the distributed model of semi-lattices and the incremental view-maintenance model of rings have seen significant attention in building prototype systems such as the Anna distributed key-value store [30, 31] and the DBToaster Incremental view maintenance system [18].

Our research group at Berkeley is building an optimizer for distributed systems [9] and in the interest of building the best optimizer possible, we are concerned with incorporating the learnings from both of these lines of work into a single system. Initially, these two different approaches seemed mutually exclusive. Lattices grow monotonically while groups and rings require an inverse operation that is provably non-monotone. This paper is about how to fit these two puzzle pieces together.

The solution is not to try to get them to work together in one structure, but to separate them out into different layers that operate independently and only interact through a translation layer. This layered approach lines up with the separation of concerns that each structure is meant to deal with in a distributed system. The role of CRDTs and lattices is to ensure robustness to nondeterminism on our asynchronous network. The group and ring approaches were designed for single-node systems and play no role in the network layer of our system. By separating these, we get the benefits of the lattice at the network layer and the benefits of the groups and rings at the query processing layer.

The remainder of this paper is organized as follows: In Section 2, we give background on semi-lattices (CRDTs) for strong eventual consistency and on the use of abelian groups in incremental view maintenance. In Section 3, we show why the strawman of combining groups and lattices into one structure fails and then give two constructions for the cohabitation of semi-lattices and abelian groups. In Section 4, we give background on rings in IVM. In Section 5, we discuss the different deletion semantics used in IVM and CRDTs. In Section 6, we discuss future work on algebra-aware data systems. In Section 7, we discuss related work, particularly op-based CRDTs and δ-CRDTs. Appendix A gives background on abstract algebra and Appendix B shows why a strawman for combining rings and lattices into one structure fails.

ACM ISBN 979-8-4007-0544-1/24/04.
https://doi.org/10.1145/3642976.3653031

2 Background

In this section, we give necessary background on semi-lattices for CRDTs and abelian groups for incremental view maintenance. Definitions of mathematical terms from abstract algebra are provided in Appendix A. Feel free to skip these sections if you are already familiar with these topics.

2.1 Semi-Lattices and Conflict-Free Replicated Data Types

Conflict-free replicated data types (CRDTs) are a popular interface for designing systems with strong eventual consistency of writes. In simple terms, this consistency guarantee means that all replicas in a system will converge to the same state as long as all updates eventually propagate to them. The interface that CRDTs provide is an object-oriented one with roots in abstract algebra. A *state based CRDT* requires that a user defines two operations on their state: an *update* operation for changes to the state from outside the system (e.g. from users), and a *merge* operation for replicas to combine the updates they have seen into one state. By satisfying certain algebraic properties in the selection of update and merge operations, a developer is guaranteed strong eventual consistency of the replica state via gossip communication over an asynchronous computer network [26, 27].

The requirements on the CRDT operations are that merge is *associative*, *commutative*, and *idempotent* and that update is *monotone* with respect to the ordering induced by the merge operation (see Appendix A for definitions). An algebraic structure that is associative, commutative, and idempotent is called a semi-lattice, so the state-based CRDT model offers us an algebraic lens on eventual consistency in distributed systems.

The reason for these three requirements on the merge operation of CRDTs is that they each provide protection against different sources of nondeterminism on the communication network. Associativity gives the system robustness to arbitrary batching, commutativity gives robustness to different interleaving or reordering of messages on the network, and idempotence gives robustness to messages being delivered multiple times or being "retried". The power of this robustness is that without coordination replicas can process updates and guarantee eventual convergence. This allows geo-replicated systems to service requests with local latency without sacrificing availability or consistency under network partitions (circumventing the CAP Theorem [5]). Note that all of the problems CRDTs are solving revolve around problems with asynchronous network communication.

A common example of a CRDT is a set. Merge is performed via set union which is associative, commutative, and idempotent. The update operation is also set union, so an update can add an element to the set and all replicas will eventually converge to exactly the set of individual updates applied across the replicas.

An alternative view of CRDTs that is common in the literature is operation-based CRDTs [26, 27]. We discuss the connections between IVM and operation-based CRDTs in Section 6, but they are not the focus of this paper, as they are not based on semi-lattices.

2.2 Groups and Incremental View Maintenance

Incremental view maintenance (IVM) is the study of how to efficiently maintain query answers over a database as the contents of the database gets updated over time. They have been studied in academia for decades and their techniques have been commercialized in a number of database systems including Materialize [19], Feldera [7], Azure Synapse [28], Amazon Redshift [4], and Databricks [22]. They continue to be a popular topic of academic research in the database community [12, 16, 18, 21] and were the topic of the 2023 VLDB best paper award [8]. For an excellent survey of incremental view maintenance see [10].

For simplicity of explanation, we start with the group-theoretic model of incremental view maintenance used in DBSP [8]. In Section 4, we extend our exploration to the ring-theoretic model which simply adds a second operator to the group to express more complex queries.

There are two common semantics for databases, the "set semantics" in which every tuple occurs at most once and the "bag semantics" in which tuples can occur multiple times [1]. The former gives us a data model in which the database is a set of tables and each table is a set of tuples. The latter gives us a data model in which the database is a set of tables but each table is a multi-set of tuples. Each tuple is "tagged" with a counter indicating its multiplicity. DBSP utilizes a generalization of these two models to enable positive *or negative* multiplicities of tuples which is called a "Z-set" [8, 14] because each tuple has a multiplicity from the integers ($\mathbb{Z}$). The benefit of allowing negative multiplicities is that we can now talk about insertions of tuples and deletions of tuples from the database in a unified way. A deletion of a tuple is simply its insertion with a multiplicity of negative one. A modification of a tuple is the deletion of that tuple followed by the insertion of the modified value.

Considering our state to be Z-sets and our incoming updates to be Z-sets as well (batches of tuples being inserted or deleted), we see that the update operation forms an abelian group. That is, our update is associative, commutative, and every incoming Z-set update, u, has a corresponding "inverse" Z-set, u^{-1}, such that update$(u, u^{-1}) = 0$.

With this model of state and updates to state, we can define operators over these Z-sets and queries composed out of operators. Given certain properties on the operators, we are able to guarantee the "incremental" computation of query results: that is, work done to compute the result after a new update can be proportional to the size of that update rather than to the size of the entire database.

The essential property of an operator that makes it efficiently incrementalizable is that it is *linear* which is defined as $f(a+b) = f(a)+f(b)$ where + is the group operator. This is equivalent to saying that operator f is a homomorphism over our group. We can see that if an operator is linear then we can compute this operator incrementally (we already have $f(old\_db\_state)$ and when a new update comes in we just do $f(new\_update) + f(old\_db\_state)$).

It turns out many useful operators are linear with respect to this Z-set abelian group such as selection and projection in database queries. Linearity is also composable, so any query we can construct as a composition of linear operators will be efficiently incrementalizable.

Another key category of operators is *bilinear operators* which are binary operators $f(a, b)$ that satisfy distributivity over +. The classic example of a bilinear operator in database queries is a join. We can think of bilinear operators as functions that would normally cost N^2 time in the database instance size to compute, but in the incremental setting we can compute them in time (update_size × N). Intuitively, when a new tuple arrives on one input to the join we need to check it against each existing tuple received on the other input once to compute the join.

It turns out that quite expressive query languages from databases are entirely incrementalizable in this model including relational algebra, Datalog, grouping, and aggregation– i.e. much of SQL and beyond. [8].

This abelian group with linear operators model offers us considerable power, but is it compatible with eventual consistency? In the next section, we give a construction for combining group-theoretic incremental view maintenance and lattice-theoretic eventual consistency.

3 Co-habitation of Abelian Groups and Semi-lattices

When we first wanted to combine these two structures, we asked the seemingly obvious question "Can the CRDT update or CRDT merge operation be the abelian group update operation?" First we will show why these two strawman approaches cannot work and then dive into our multi-layer solution.

Strawman 1: CRDT Update as Group Update: The update operation of a CRDT must grow monotonically with respect to some partial order. An abelian group update operation must have an inverse update operation. For both of these to be true, the group can only have one element (rendering it useless for expressing application semantics):

Proof: Assume + is monotonically non-decreasing ($\forall x, y : x + y \geq x$. Then $x + -x \geq x$ and $-x + x \geq x$. Adding x and -x to the respective sides we get $x \geq 0$ and $-x \geq 0$. We know $x + -x = 0$ so $0 \geq x$ and $0 \geq -x$. So $x = -x = 0$ for every x in the group.

Strawman 2: CRDT Merge as Group Update: Recall that a CRDT merge operation must be idempotent. We show that if an abelian group operation is idempotent, then the group must also be the one element group $(\{0\}, +)$.

Proof: For any idempotent element x in the group G, we have that $x+x = x$. Due to invertibility of +, x must also have an additive inverse, $(-x)$. Adding this to both sides yields $x = 0$, meaning all elements must be the additive identity. Thus, with an idempotent + operation, G must be the one element group $(\{0\}, +)$.

Powering through the disappointment of these failed strawmen, we find that there is still a way for CRDTs and IVM groups to co-habitate! The trick is in separating the CRDT from the group and adding a translation layer that allows these two structures to co-exist. We first give a working construction for this using a simple set CRDT that is inefficient but demonstrates how the group and semi-lattice combine. In the next section, we provide a variant with a performance-optimized CRDT based on delta-CRDTs [29].

The key observation to see how group and semi-lattice structures can be combined is to think about how these two structures are really being used in our data system. We have some state of our data. We want to modify that state in an incremental way. For that, we need this + operation that forms an abelian group. We can then express dataflow queries with linear and bilinear operators over that + operation.

Then what are lattices for in our data-intensive systems? **The role of the lattice is to allow us to replicate state while being protected against nondeterminism of computer networks.** The network plays no role in our data modification (+) or query evaluation. The network is a different layer of the system. Much like we write our application semantics without concern for how TCP is being used for delivery, we can write our group-based or ring-based application without concern for how our lattice is handling network nondeterminism. Much like with TCP, we leave our application alone with its + operation, and then at a lower layer we propagate updates around the system wrapped up in a nice robust semi-lattice. We call this construction a "lattice-wrapper" and depict this idea visually in Figure 1.

3.1 The Very Simple Construction

The pseudocode for this construction is given in Listing 1. At a high level, input updates arrive of type Z-set to apply to the local group structure at a replica. We pass the Z-set value into the group to modify the group state and update the materialized views at that replica. We also convert this incoming Z-set value into a semi-lattice value by pairing it with a randomly generated unique ID. The (updateID, Z-set) pair is propagated to other replicas which process this update by keeping track of the set of updateIDs they have seen and ignoring any repeated updateIDs. When the receiver hasn't seen the incoming updateID before, it adds it to its list of seen updates and passes the Z-set payload into its local group to be processed. Our lattice state is a set of updateIDs which we modify (merge) via set union - an associative, commutative, and idempotent operation.

Note that in the design that we have described, we do not enforce that updates arrive in FIFO order or causal order. It is only a couple extra lines of code in our merge function to do this (we wait until we see the updates in-order from the sender before passing them into the group), but because our update operation is an abelian group it is commutative, so we don't need to bother with any extra metadata or logic for enforcing update ordering. This commutativity of the group frees us from the worry of ordering; eventual consistency and linearizability give the same results at the application level for abelian group-based and ring-based applications.

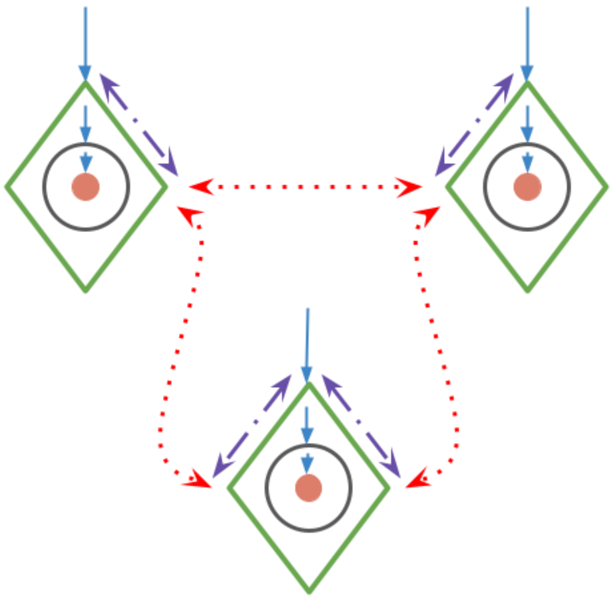

Figure 1. We depict three replicas of our lattice-wrapped view groups. Blue solid arrows are the incoming updates to the database instance of type Z-set. Red dotted arrows are CRDT merge operations being broadcast to each replica. The green diamonds represent the lattice wrappers and we see the solid blue updates are converted into dotted red merges via the lattice wrapper. The conversion is depicted by a purple semi-dotted arrow. We see that the updates are passed through to the black group (circle) inside the lattice wrapper. The red dot in the center is the materialized view that users can observe and where the dataflow pours into. We see that merge operations are received by the lattice wrappers and converted into update operations that are then passed through to the inner group structure.

3.2 The Performant Construction

The simple construction above requires each node to keep track of the list of every updateID they have ever seen. To

Listing 1. Pseudocode for Z-set lattice wrapper construction

```
let mut my_group = Group::Zset::new();
let mut my_inbox = Lattice::Set::new();

fn processUpdate(&mut self, update_payload: &Zset)
    {
        self.my_group.apply(update_payload);
        let update_id = Uuid::new_v4();
        let update_wrapper = (update_id,
            update_payload);

        my_outbox.insert(update_wrapper);
        neighbors.send(my_outbox);
}

fn receiveUpdate(&mut self, (incoming_uuid,
    incoming_zset): (Uuid, &Zset)) {
        if (!my_inbox.contains(incoming_uuid)) {
            self.my_group.apply(incoming_zset);
            my_inbox.insert(incoming_uuid);
        }
}
```

improve on this metadata overhead, we give a construction in this section that reduces this metadata to be proportional to the number of replicas on average rather than the number of updates in the system.

The lattice wrapper we use is similar to the delta-CRDT lattice wrapper [29] and familiar in networking literature as a part of the TCP protocol. In English, each replica has a uniqueID and a local logical clock that it increments each time it receives a local update to the group. It uses this (uniqueID, localLogicalClock) pair as the updateID key to the Z-set update payload. It sends the element of the map resulting from each update along to the other nodes in the system. A replica receiving one of these map elements processes it using the merge operation, which like before simply checks whether that (uniqueID, localLogicalClock) is already in the set of updates the receiving node has processed. To reduce metadata overheads, the sequence of clock times per node can be stored in heavily compressed representations as they are long runs of contiguous integer values. The receiving nodes will also acknowledge received updates to the sender so the sender can garbage collect its map of update payloads once each other replica has acknowledged hearing about that update payload. The sender will re-send updates to nodes that haven't acknowledged their receipt after a fixed time interval. This is like the construction used in [29] and you can see that paper for more details.

4 Rings in Incremental View Maintenance

One of the major successes of databases is the power of query optimizers which take a logical query plan (similar to an abstract syntax tree) and search the space of equivalent dataflow graphs (physical query plans) for the one that will have the best performance. The space of equivalent dataflow graphs is defined in terms of different rewrite rules that can be applied to the logical query plan without changing the result of the query. These rewrite rules can be seen as algebraic axioms on the operators in the query language.

Common optimization techniques include choosing what order to execute joins in (associativity of join), choosing what order to apply filters in (commutativity of selection), and choosing whether to push filters before joins in the query plan (distributivity of filters over join). One observation is that the standard axioms that optimizers leverage for relational algebra form a semi-ring. A semi-ring at a high-level is a structure with two operators where one is associative and commutative, the other is just associative, and the second operator distributes over the first operator (see Appendix for full definition). With this observation, database researchers have generalized the applicability of database techniques beyond relational algebra to any pair of operators that satisfy the algebraic axioms of a semi-ring. This semi-ring perspective has been popular recently in databases for studying data provenance [14], recursive queries [2], and incremental view maintenance [18].

The reason that we focus on rings instead of semi-rings in incremental view maintenance is because rings have inverses on +, and + corresponds to the update operation on the data. Thus, the + in a ring forms an abelian group and gives us the nice properties we discussed in Section 2. The × operator in the ring is analogous to *join* in relational algebra.

How does this picture of incremental view maintenance as a ring instead of as a group change our picture and lattice wrapper construction? The answer is that it doesn't need to change our lattice wrapper at all as the group and lattice structures already operate independently. The ring just means that the queries we express over the state of the group can be optimized with rewrite rules and get more performant dataflow graphs for the computation of views inside our group.

5 Inverses, Two-Phase Sets, and the Semantics of Deletion

In this section, we take a brief detour to discuss the interesting differences in deletion semantics found in Z-sets compared to CRDTs. We do not conclude that one semantics is better or worse than another; we simply highlight the differences and observe that some applications would prefer one and some would prefer the other.

Deletion semantics in **CRDTs** have been a long-standing challenge. In order to satisfy the *monotone update* requirement, complex lattice structures have been introduced offering different tradeoffs for deletion semantics. The two-phase set design [23], for example, circumvents monotone updates, but runs into two other problems with deletion; the "natural" single-node semantics of deletion are neither commutative nor idempotent. Commutativity means the order of operations in a sequence doesn't change the outcome, but if we think about "insert A; delete A" vs "delete A; insert A" the common meaning of this on a single node interface would be that the result of the first operation sequence is the empty set and the result of the second sequence is the set $\{A\}$. This non-commutativity leads the two-phase set design to treat all deletions as if they occur after all insertions - a decision that is quite unsatisfactory in the simulation of a single-node user experience.

Another problem with this two-phase set design is that in a common interface it should be possible to insert something, remove it, and then insert it again. Under the idempotent interpretation of deletion used in the two-phase set this is not possible; the item can be inserted at most once and deleted at most once. This deletion with *at most once* semantics is often referred to as "tombstoning".

To resolve these awkward semantics, the observe-remove set (OR-set) [24] additionally tracks the causality of updates to fix the idempotent deletions. The causality also fixes the single-node experience of "delete A; insert A", but for concurrent writes ambiguity in semantics remains, so a user must pick whether to default to deletes first or inserts first.

Incremental view maintenance literature has focused on the single-node setting and in that setting they are able to take a very simple and clean view of deletions. We treat the database instance as a Z-set under the hood. An insertion or deletion from the database increments or decrements the multiplicity of the specified tuple. For user-facing multiset semantics we may display all negative multiplicities as 0. For a CRDT-view of this we can think of each tuple as having a pn-counter as one of its columns representing the multiplicity of the tuple.

The Z-set semantics itself is not a valid semantics for a state-based CRDT as the update operation is not monotone and the state changes in non-idempotent ways, but we know that those concerns can be handled in our lattice wrapper layer, so what about the deletion semantics themselves? For Z-sets, insertions and deletions are fully commutative, so if a user issues a delete and expects the count to go from 0 to 0 then this will be broken. Fully commutative updates also lead to the (insert A; delete A) vs (delete A; insert A) anomaly from two-phase sets. However, prevention of both of these scenarios can be handled at the client by ignoring deletions when the observable count at the client is 0. The "observable count at the client" is exactly the updates that occur causally before the new update at that client.

One anomaly that can still occur is if two people see there is a count of one and simultaneously decide to delete it. In Z-set semantics this would result in a count of -1 if the deletions both occur within the gossip time window ("concurrently"). The best semantics for such a case is application specific, but the Z-set construction with client-side guards avoids tombstoning and achieves something that could reasonably be called "causal multiset semantics".

6 Discussion and Future Work

Exploring these two algebraic lenses allows us to get the best of both worlds in our systems. Our research group is interested in building algebra-aware systems in which the algebraic properties of application logic are known and can be utilized by the system as much as possible. In this paper, we began this journey by connecting just two of the many algebraic optimizations known in computer science. In future work, we plan to draw connections to the semi-ring approaches to provenance [14] and fixed point computation [2] from databases as well as to areas of computer science beyond databases. Algebra has played a central role in modern cryptography including the use of group and rings for public key cryptography [25] and the use of ideal lattices for fully homomorphic encryption [13].[1] Algebra also shows up in program analysis [6] as well as high-performance computing [17, 34]. Once we have a system that understands algebraic properties, we hope to apply the learnings from these other fields to it as well.

We have started this work using semi-lattices and have released a Rust *lattices crate* (https://hydro-project.github.io/hydroflow/book/lattices_crate.html) that gives a simple interface for developers to fuzz test whether their custom lattices satisfy the necessary algebraic properties. We are working on extending this crate to support rings, groups, fields, and semi-rings to make building systems out of these primitives more ergonomic for developers.

7 Related Work

Our study of the co-habitation of semi-lattices and rings is inspired by recent work in **incremental view maintenance** such as DBSP [8], DBToaster [18], and Differential Dataflow [20]. DBSP takes a group-theoretic view of updates and focuses on single-node updates without query optimizations. DBToaster was another single-node system based on rings and supporting traditional relational query workloads with optimized and incremental updates and queries. Differential Dataflow is an incremental computation framework that considers the distributed case, although not specifically in the context of eventual consistency.

To our knowledge, we are the first work to explicitly connect the algebraic view of incremental view maintenance from the databases literature and the algebraic view of eventual consistency from the distributed systems literature. Other works have explored distributed IVM in the context of transactions [15] and data warehousing [3, 32, 33].

We have focused our attention in this paper on state-based CRDTs rather than op-based CRDTs. The reason for this is that state-based CRDTs are modeled naturally as algebraic semi-lattices and their relationship to rings is interesting. Op-based CRDTs can be thought of as delegating the semi-lattice properties to a networking layer and dealing only with the application layer properties. In a sense, this makes the op-based CRDT perspective and the co-habitating rings and semi-lattices perspective similar - the mechanisms for ensuring exactly once delivery over the network can be treated separately from the mechanisms for managing changes to application-visible data.

Op-based CRDTs require that the update operation be associative and commutative, forming a commutative monoid. This is much like an abelian group except it drops the requirement that updates have inverses. Some existing op-based CRDTs like counters [24] already have an inverse operation, forming an abelian group and being amenable to DBSP-style incrementalization. Other op-based CRDTs like set CRDTs do not support inverses, but they raise the interesting question of what classes of operators and queries over commutative monoids are automatically incrementalizable. The definitions of linearity and bilinearity are the same for commutative monoids as for abelian groups. We leave a formal treatment of this connection to future work.

δ-CRDTs [29] are a design that minimizes the network overheads of state-based CRDT communication. Much like IVM, they make the network utilization proportional to the size of an update batch rather than proportional to the size of the state.

8 Conclusion

We have presented a way to utilize two different algebraic views of data systems, CRDTs and incremental view maintenance, in the same holistic system. The seemingly incompatible semi-lattice and ring structures can be made to cohabitate by using ring structures at the "application layer" and semi-lattices at the "network layer". With this, we are able to perform incremental computation of complex query plans while guaranteeing coordination-free strong eventual consistency.

9 Acknowledgements

This work was supported by gifts from Accenture, AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, SAP, Uber, Meta, and VMware. Hellerstein was supported in part by Sutter Hill Ventures. We thank Val Tannen for his helpful feedback and Jasmine Macasaet for her help with the design of Figure 1.

[1]Different type of lattice than the semi-lattice used in CRDTs.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.

[2] Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, Dan Suciu, and Yisu Remy Wang. 2022. Convergence of datalog over (pre-) semirings. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 105–117.

[3] Divyakant Agrawal, Amr El Abbadi, Achour Mostéfaoui, Michel Raynal, and Matthieu Roy. 2002. The lord of the rings: Efficient maintenance of views at data warehouses. In *Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16*. Springer, 33–47.

[4] AWS. 2021. Refreshing a Materialized View. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh.html.

[5] Eric Brewer. 2012. CAP twelve years later: How the" rules" have changed. *Computer* 45, 2 (2012), 23–29.

[6] Manfred Broy, Martin Wirsing, and Peter Pepper. 1987. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 1 (1987), 54–99.

[7] Mihai Budiu. 2024. Incremental Database Computations. https://www.feldera.com/blog/incremental-database-computations/.

[8] Mihai Budiu, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2022. DBSP: Automatic incremental view maintenance for rich query languages. *arXiv preprint arXiv:2203.16684* (2022).

[9] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. 2021. New directions in cloud programming. *arXiv preprint arXiv:2101.01159* (2021).

[10] Rada Chirkova, Jun Yang, et al. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[11] David Steven Dummit and Richard M Foote. 2004. *Abstract algebra*. Vol. 3. Wiley Hoboken.

[12] Iman Elghandour, Ahmet Kara, Dan Olteanu, and Stijn Vansummeren. 2018. Incremental techniques for large-scale dynamic query processing. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2297–2298.

[13] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[14] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[15] Richard Hull and Gang Zhou. 1996. A framework for supporting data integration using the materialized and virtual approaches. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 481–492.

[16] Ahmet Kara, Hung Q Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2020. Maintaining triangle queries under updates. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–46.

[17] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[18] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. 2014. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *The VLDB Journal* 23 (2014), 253–278.

[19] Jessica Laughlin. 2020. Why use a Materialized View? https://materialize.com/blog/why-use-a-materialized-view/.

[20] Derek G Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, iterative data processing with timely dataflow. *Commun. ACM* 59, 10 (2016), 75–83.

[21] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. 2020. F-IVM: learning over fast-evolving relational data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2773–2776.

[22] Yannis Papakonstantinou Nitin Sharma Andreas Neumann Paul Lappas, Michael Armbrust. 2023. Introducing Materialized Views and Streaming Tables for Databricks SQL. https://www.databricks.com/blog/introducing-materialized-views-and-streaming-tables-databricks-sql.

[23] Nuno Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. *arXiv e-prints* (2018), arXiv–1806.

[24] Nuno Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free replicated data types (CRDTs). *arXiv preprint arXiv:1805.06358* (2018).

[25] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[26] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of convergent and commutative replicated data types. In *Technical Report, Inria–Centre Paris-Rocquencourt; INRIA*.

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

[28] Sindhu Subhas. 2021. Hive - Materialized Views. https://techcommunity.microsoft.com/t5/analytics-on-azure-blog/hive-materialized-views/ba-p/2502785.

[29] Albert van der Linde, João Leitão, and Nuno Preguiça. 2016. δ-crdts: Making δ-crdts delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. 1–4.

[30] Chenggang Wu, Jose M Faleiro, Yihan Lin, and Joseph M Hellerstein. 2019. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* 33, 2 (2019), 344–358.

[31] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment* 12, 6 (2019), 624–638.

[32] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. 1995. View maintenance in a warehousing environment. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 316–327.

[33] Yue Zhuge, Hector Garcia-Molina, and Janet L Wiener. 1996. The strobe algorithms for multi-source warehouse consistency. In *Fourth International Conference on Parallel and Distributed Information Systems*. IEEE, 146–157.

[34] Ling Zhuo and Viktor K Prasanna. 2005. High performance linear algebra operations on reconfigurable systems. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE, 2–2.

A Abstract Algebra Background

A *semi-lattice* is an set L equipped with a binary operation $\sqcup$ that satisfies three properties: associativity, commutativity, and idempotence.

associativity: $\forall a, b, c \in L : (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
commutativity: $\forall a, b \in L : a \sqcup b = b \sqcup a$
idempotence: $\forall a \in L : a \sqcup a = a$

Any valid semi-lattice operator induces the following partial ordering on the set L: $a \sqcup b = b \rightarrow a \leq b$.

A *monotonically non-decreasing* operation, update(), with respect to a partial ordering $\leq$ satisfies $\forall a \in L : a \leq update(a)$

A *group* is a set G equipped with a binary operation $+$ that satisfies two properties: associativity and invertibility.

A group is also required to have an identity element, 0, satisfying $\forall g \in G : 0 + g = g + 0 = g$.

invertibility: $\forall g \in G \exists - g \in G : g + (-g) = 0$

A *abelian group* is simply a group where + is also commutative.

A *ring* is a set S equipped with two binary operations + and × satisfying the following algebraic properties: + is associative, commutative, has an additive identity, and has additive inverses, × is associative and has a multiplicative identity, and × is distributive over + (both left-distributive and right-distributive). On their own, the + operation forms an *abelian group* over S and the × operation forms a *monoid* over S.

Formalizing the terms not already defined above:

+ has an *additive identity* if $\exists 0 \in S : \forall a \in S : a + 0 = a$. We call this element "0" the additive identity.

+ has an *additive inverse* if $\forall a \in S : \exists b \in S : a + b = 0$. We say b is the additive inverse of a in this case, and will often denote this element b as a^{-1}.

× has a *multiplicative identity* if $\exists 1 \in S : \forall a \in S : a \times 1 = 1 \times a = a$. We call this element "1" the multiplicative identity.

× is *left-distributive* over + if $\forall a, b, c \in S : a \times (b + c) = (a \times b) + (a \times c)$

× is *right-distributive* over + if $\forall a, b, c \in S$: $(b + c) \times a = (b \times a) + (c \times a)$

a *semi-ring* is just a ring where the + operation does not need to be invertible.

a map φ from one ring R to another ring S is a *ring homomorphism* if it satisfies: (1) $\varphi(a + b) = \varphi(a) + \varphi(b)$, (2) $\varphi(ab) = \varphi(a)\varphi(b)$, (3) $varphi(1) = 1$, and (4) $varphi(0) = 0$.

a ring homomorphism φ is a *ring isomorphism* if and only if φ is bijective. Two rings are *isomorphic* if and only if there exists a ring homomorphism between them.

B Impossibility of Strawman Approaches for Rings

If the CRDT operations do not work in an abelian group, is it possible to have them work in a ring? We quickly illustrate why this simple approach does not work either. In section 3, we showed the problems with combining an abelian group with a CRDT. Since the + operation of a ring forms an abelian group, the arguments in section 3 already disallow the possibility of either CRDT merge or CRDT update being + in our ring. That leaves ×.

The main issue with × being the CRDT merge operation is that × would then have to be idempotent, after which a crucial theorem from abstract algebra about idempotent rings gets in our way. This theorem can be found in standard abstract algebra textbooks [11].

Theorem Any ring with an idempotent × operation is isomorphic to many copies of the Boolean ring, ({0,1}, XOR, AND).[2]

Proof: For any element e in the ring R idempotent under ×, we can form a map $\varphi : R \longrightarrow Re \times R(1 - e)$, defined by $\varphi(x) = (xe, x(1 - e))$. It can be easily proven that φ is a ring isomorphism, so that for any two elements x, y in R, $\varphi(xy) = \varphi(x)\varphi(y)$, and $\varphi(x + y) = \varphi(x) + \varphi(y)$. Thus, if all elements are idempotent, choose any element x, so that $R \cong Rx \times R(1 - x)$. Since $Rx, R(1 - x)$ are subrings of R, we can decompose both of them into two subrings of their own and repeat the process. This decomposition continues until each subring only has two elements, namely 0 and 1. Giving these subrings an associative, commutative, invertible + operation and an associative, commutative, idempotent × operation yields the Boolean ring $(\{0, 1\}, XOR, AND)$.

If the × operation in our ring is a valid CRDT merge operation, then this forces + to be the XOR operation on $\{0, 1\}$. However, XOR is not monotone with respect to the partial ordering induced by × $(1 \leq 0)$, and therefore, + cannot be the CRDT update operation.

[2]The Boolean ring is better known as $\mathbb{Z}/2\mathbb{Z}$, the integers modulo 2. The XOR and AND operations are equivalent to + and × operations in the integers mod 2.