SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Expressing CRDTs with Datalog

Leo Stewen

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Expressing CRDTs with Datalog

Definieren von CRDTs mit Datalog

Author:	Leo Stewen
Examiner:	Prof. Dr. Viktor Leis
Supervisor:	Dr. Martin Kleppmann
Submission Date:	17.07.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 17.07.2025 Leo Stewen

Acknowledgments

To my father, rest in peace. I feel endless gratitude for having had you in my life and for the path we shared.

I am grateful for the support of my supervisor, Dr. Martin Kleppmann, from whom I learned a great deal about distributed systems and academic writing. Thank you for critically reflecting my work and offering me your excellent feedback. Thank you also for your patience with and trust in me.

I would like to express my gratitude to the creators of the Rust programming language for their efforts in making systems programming more accessible and sane, as well as to the creators of Zed for developing an outstanding editor.

Finally, I am grateful to Prof. Dr. Viktor Leis for providing formal supervision and examination on behalf of TUM.

Abstract

CRDTs are data structures which allow replicas to converge to the same state in the presence of uncoordinated, concurrent writes. The latter make it challenging to define CRDTs correctly and complicate the proof of their convergence. To eliminate an entire class of convergence-related errors, researchers have proposed defining CRDTs in restricted, domain-specific languages that ensure CRDTs always converge.

This work explores using Datalog — applied over a monotonically growing set of operations — as a domain-specific language for defining CRDTs with guaranteed convergence. To avoid performance degradation as the set of operations grows, I utilize incremental computation via the recent DBSP framework to evaluate Datalog CRDT queries on a custom-built query engine. Its performance is evaluated using a key-value store and a list CRDT.

This work is an initial step towards a future in which application developers are empowered to define their own CRDTs in a high-level query language without needing to worry about their concrete implementation nor their convergence. More broadly, it contributes to the ongoing effort to understand the performance cost of guaranteed convergence.

Kurzfassung

CRDTs sind Datenstrukturen, die es Replikaten ermöglichen, zum gleichen Zustand zu konvergieren, trotz unkoordinierter, nebenläufiger Schreibzugriffe. Letztere erschweren die korrekte Definition von CRDTs und machen den Beweis ihrer Konvergenz kompliziert. Um eine ganze Klasse konvergenzbezogener Fehler zu eliminieren, haben Forscher vorgeschlagen, CRDTs in eingeschränkten, domänenspezifischen Sprachen zu definieren, die ihre Konvergenz garantieren.

Diese Arbeit untersucht die Verwendung von Datalog — angewendet auf eine monoton wachsende Menge von Operationen — als domänenspezifische Sprache zur Definition von CRDTs mit garantierter Konvergenz. Um einer Leistungsverschlechterung bei wachsender Operationsmenge entgegenzuwirken, wird inkrementelle Berechnung mit Hilfe des DBSP-Frameworks benutzt, um Datalog-CRDT-Abfragen auf einer eigens entwickelten Abfrage-Engine auszuführen. Die Performance wird mithilfe eines Key-Value-Stores und einer Listen-CRDT evaluiert.

Diese Arbeit stellt einen ersten Schritt für eine Zukunft dar, in der Anwendungsentwickler ihre eigenen CRDTs in einer Abfragesprache definieren können, ohne sich um deren konkrete Implementierung oder Konvergenzeigenschaften sorgen zu müssen. Im weiteren Sinne trägt sie zur laufenden Erforschung der Kosten garantierter Konvergenz bei.

Contents

1 Introduction

Convergent replicated data types (CRDTs) are a class of data structures that allow replicas in a distributed system to converge to a consistent shared state without any coordination of reads and writes to the data structure, i.e., they can be served from the local state of a replica. This allows for offline writes at each replica. In case of concurrent writes, replicas are guaranteed to converge to the same state, once they have exchanged their writes with each other. The convergence guarantee is what renders defining and implementing correct CRDTs challenging. Currently, application developers wanting to use CRDTs in their applications face an unfortunate trade-off: Either they use an existing CRDT library [1]–[3] that is developed by experts in the field using a general-purpose programming language but exposes a fixed, object-oriented API for a fixed set of CRDTs, or they define their own CRDTs tailored to their specific use case. If they opt for the latter, they either have to invest significant efforts into implementing it and proving that the CRDT algorithm (as well as its implementation) is commutative when applying concurrent writes, which is a labor-intensive, subtle, and error-prone task [4], [5], requiring expertise in formal verification. Alternatively, they can learn a domain-specific language (DSL), like VeriFx [6] or LoRe [3], which have a restricted expressiveness but allow CRDTs to be defined in a way that guarantees their convergence under concurrent writes. Yet, restricted DSLs are not always expressive enough to define and verify the desired CRDT: For instance, VeriFx can verify a fractional-indexing list CRDT but the verification of the RGA list CRDT, which I define in Datalog in Section 4.1.2, times out [6]. Fractional-indexing uses arbitrary precision rational numbers for the list elements' indices, and insertions pick a value between its predecessor and successor elements as their index. While this is verifiable, it is inefficient in practice and causes interleaving issues of concurrent insertions from different replicas [7].

Previous work [8] has proposed Datalog as a DSL to express CRDTs as queries over a monotonically growing set of immutable operations. The set of immutable operations can easily be disseminated among replicas by gossip, or other protocols such as set reconciliation [9], [10]. Such protocols are well suited for replication in decentralized peer-to-peer systems. Crucially, however, is Datalog's deterministic execution upon sets (or multisets), which ensure that queries cannot rely on the processing order of operations, guaranteeing convergence for replicas executing it on the same set of operations, as I lay out in more detail in Chapter 2. With this approach, application developers do not have to worry about the convergence property of their custom CRDT anymore, as it is guaranteed by construction, and are empowered to design their own CRDTs. While still being a restricted language, Datalog may be more expressive and better known than a custom DSL. Yet, so far the idea has only been hypothesized. This work aims to put the idea into practice by exploring the feasibility of expressing CRDTs as Datalog queries through:

- **Implementing a query engine** which can execute queries represented in an intermediate representation (Section 3.1). The intermediate representation works on top of operators from relational algebra, e.g., joins, projections, and selections. The engine uses incremental view maintenance, by leveraging the recently released DBSP framework [11], to incrementally maintain the state of a query over a stream of updates.
- **Defining and implementing a Datalog dialect** (Section 3.2), powerful enough to express CRDTs as queries. The dialect includes support for self-recursion and negation and is translated to the query engine's intermediate representation (Section 3.3).
- **Providing CRDT definitions** of a key-value store (Sections 1.1 and 4.1.1), a list (Section 4.1.2), and a causal broadcast (Sections 1.2 and 4.1.1) in my Datalog dialect. Section 4.2 evaluates their performance in two settings: (1) simulating an application restart by feeding in all updates to the CRDT at once, and (2) simulating near-real-time collaboration by feeding in recent updates on top of an existing state.

Ideally, this different approach to CRDTs may move them closer to the power, guarantees and flexibility of database systems. The higher abstraction level of a query language better supports the decoupling from logical and physical data representations than object-oriented APIs. Additionally, advances in query optimization and performance improvements of the query engine can be introduced without breaking changes, something relational databases have benefitted from over the years and has arguably contributed towards their widespread adoption.

Conversely, database systems can be introduced to coordination-free environments, in which every replica can read and write to the database without coordinating with other replicas. In exchange for some guarantees, which cannot be upheld in such environments, e.g. primary key constraints [12], coordination-free database systems can offer ultimate availability: As long as at least one replica is alive, the database remains available. This property can be useful in contexts where a system must be resilient against network partitions. For instance, in manufacturing, an entire production line should not be stopped just because some server is not available. Although CRDTs are mostly discussed in the context of collaborative applications, they are not limited to this domain: Companies like Ditto [13] use CRDTs for applications in manufacturing, aviation, gastronomy and military sectors.

Incremental view maintenance allows queries to be computed incrementally, i.e., it takes input *changes* and outputs *changes* to the query's result. Receiving output changes instead of the full state provides more information to the application than just the final state of the query. That way, the idea of functional reactive programming can be extended from the GUI to the whole application stack, including the database layer. Then, the application can be considered a reactive, pure function of some base state [14] and the output changes can be used to inform the GUI which views require rerendering. Furthermore, showing data changes, e.g. diffs, to the user is naturally supported without having to (ab)use a database's write-ahead log or similar change data capture mechanisms. Most application developers are already used to the concept of a query, as nearly every application relies on a database in some form. CRDT queries and non-CRDT queries can then share the same interface [14], reducing the cognitive load and the complexity of the application stack.

1.1 Motivating Example: A Key-Value Store as a Query

This example demonstrates how a key value store, consisting of multi-valued registers (MVRs), can be expressed with a query language. A MVR is a generalization of a last-writer-wins register (LWWR). Unlike the latter, a MVR exposes conflicting values to the application as a consequence of concurrent writes to the register. Therefore, a concurrency detection mechanism is required. For this example, I use causal histories in which every operation specifies a set of predecessor operations that it causally depends on. Version vectors are another mechanism to detect concurrency but they do not pair well with relational data models.

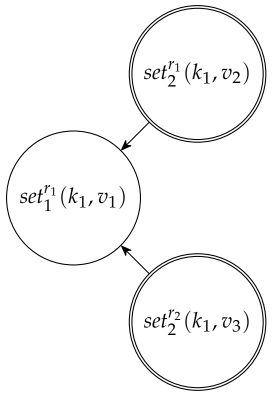
I use two relations to store the operations on the registers of the key-value store. The `set` relation contains a log of all operations that *set* the value of a register of the key-value store. The `pred` relation stores the causal dependencies between the operations. The schema of both relations and some example data is shown in Figures 1.1a and 1.1b. In operation-based CRDTs, a pair of `ReplicaId` and `Counter` values (abbreviated as `RepId` and `Ctr`, respectively) is frequently used to uniquely identify an operation. The replica id is a unique identifier for the replica that generated the operation, and the counter is essentially a Lamport clock [15]. Lamport clocks are useful to order concurrent operations on a register. Having each replica draw a replica id randomly from a sufficiently large space, such that the probability of a collision is negligible, allows replicas to generate unique identifiers without a central authority.

RepId	Ctr	Key	Value
r_1	1	k_1	v_1
r_1	2	k_1	v_2
r_2	2	k_1	v_3
...	...	...	...
r_1	3	k_2	u_1
r_2	4	k_2	u_2
r_2	5	k_2	u_3

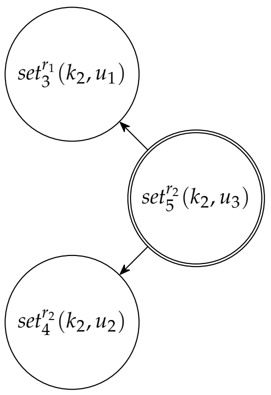
(a) `set` relation

FromRepId	FromCtr	ToRepId	ToCtr
r_1	1	r_1	2
r_1	1	r_2	2
...	...	...	...
r_1	3	r_2	5
r_2	4	r_2	5

(b) `pred` relation



(c) Causal history of register k_1



(d) Causal history of register k_2

Figure 1.1: The relations `set` and `pred` with example data (top) and their causal history illustrated (bottom).

The causal history of the operations is illustrated on a logical level in Figures 1.1c and 1.1d. The edges denote the entries of the `pred` relation and a $set_{Ctr}^{RepId}(Key, Value)$ label of a node represents a tuple of the `set` relation. To obtain the state of the key-value store, the following set must be computed:

$$\begin{aligned} mvrStore = \{(Key, Value) \mid & (RepId, Ctr, Key, Value) \in set \\ & \wedge \nexists (FromRepId, FromCtr, \_, \_) \in pred : \\ & RepId = FromRepId \wedge Ctr = FromCtr\} \end{aligned}$$

Intuitively, the query selects all key-value pairs from the `set` relation that have not been overwritten. The result is $\{(k_1, v_2), (k_1, v_3), (k_2, u_3)\}$ because other assigned values (v_1 for k_1; u_1, u_2 for k_2) have been overwritten by later operations. Figure 1.2a shows a Datalog query that computes the state of the MVR key-value store. Datalog is build around the concept of deriving new facts from existing ones by (repeatedly) applying rules which are stated in the form of implications. A Datalog query consists of a set of rules, each of which has a head (left-hand side) and a body (right-hand side), separated by ":-". If the body of a rule is satisfied for an assignment of variables, the implication is considered true for that assignment, and a new fact is derived according to the head of the rule. As Datalog is inspired by first-order logic, the Datalog query of the key-value store resembles the mathematical notation above but uses the additional "overwritten" predicate to make it more readable.

The query can also be expressed in SQL, and I demonstrate two variants. The first one in Figure 1.2b uses a `LEFT JOIN` and a `null` filter. The second one in Figure 1.2c uses a subquery and negative set inclusion, to align the SQL query closer with the mathematical notation.

```
overwritten(RepId, Ctr) :- pred(RepId, Ctr, _, _).
mvrStore(Key, Value)    :- set(RepId, Ctr, Key, Value),
                           not overwritten(RepId, Ctr).
```

(a) The MVR key-value store in Datalog.

```sql
SELECT key, value
FROM set LEFT JOIN pred ON set.RepId = pred.FromRepId
                       AND set.Ctr = pred.FromCtr
WHERE pred.FromRepId IS NULL;
```

(b) The MVR key-value store in SQL using a left join.

```sql
WITH overwritten AS (SELECT FromRepId, FromCtr FROM pred)
SELECT key, value FROM set WHERE (RepId, Ctr) NOT IN overwritten;
```

(c) The MVR key-value store in SQL using a subquery and set difference.

Figure 1.2: The MVR key-value store in Datalog and SQL.

These examples demonstrate how little code is required to express a relatively simple CRDT as a query, as opposed to hand-coding it in an imperative programming language, demonstrating another advantage of higher level abstractions. While for this example, the SQL queries are not too far off from the mathematical notation, the

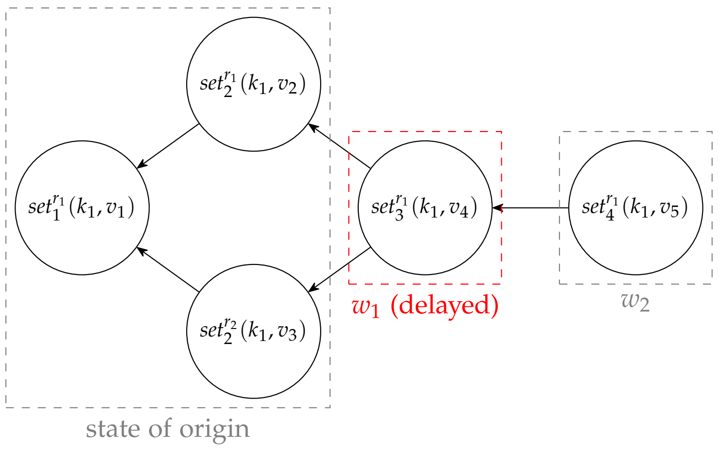


Figure 1.3: Example of a causality issue with the naive queries from Section 1.1.

next section shows that this is not always the case and motivates why I turn towards Datalog as a query language in this work.

1.2 Advanced Example: Respecting Causal Order

Causal broadcast ensures that each operation is only delivered at each replica once all of its causal dependencies have been delivered. The causal dependencies of an operation consist of all previous operations that may have causally influenced that operation, i.e., they are the set of operations whose effects were known to the operation at the moment it was created. The (partial) order of operations imposed by a causal broadcast is called *causal order*.

The example from Section 1.1 is only a correct CRDT, if the tuples in the `set` and `pred` relations are added in causal order. It produces incorrect output if updates are received out-of-order at a replica. To illustrate this issue, consider Figure 1.3 from the perspective of replica r_2 and limited to register k_1. Initially, r_2 is in the familiar state from Figure 1.1c. Then, write w_2 from replica r_1 is delivered to r_2 although its causal dependency w_1 has not been delivered yet. At this point, the query from Section 1.1 would return $\{(k_1, v_2), (k_1, v_3), (k_1, v_5)\}$ but the correct result respecting causal order is $\{(k_1, v_2), (k_1, v_3)\}$. w_2 must be buffered and only applied once w_1 has been delivered.

To prevent this, the query has to detect such "gaps" in the causal history. Hence, the problem of causal delivery is equivalent to a graph reachability problem: Which nodes are reachable from the set of root nodes, i.e., the set of nodes that are not causally dependent on any other operation. Figure 1.4a extends the query from Section 1.1 with

```
// EDBPs are omitted in this chapter.
overwritten(RepId, Ctr)     :- pred(RepId, Ctr, _, _).
overwrites(RepId, Ctr)      :- pred(_, _, RepId, Ctr).
isRoot(RepId, Ctr)          :- set(RepId, Ctr, Key, Value),
                               not overwrites(RepId, Ctr).
isLeaf(RepId, Ctr)          :- set(RepId, Ctr, Key, Value),
                               not overwritten(RepId, Ctr).
isCausallyReady(RepId, Ctr) :- isRoot(RepId, Ctr).
isCausallyReady(RepId, Ctr) :- isCausallyReady(FromRepId, FromCtr),
                               pred(FromRepId, FromCtr, RepId, Ctr).
mvrStore(Key, Value)        :- set(RepId, Ctr, Key, Value),
                               isCausallyReady(RepId, Ctr),
                               isLeaf(RepId, Ctr).
```

(a) The MVR key-value store including causal broadcast in Datalog.

```
WITH overwritten AS (SELECT FromRepId, FromCtr FROM pred)
WITH overwrites AS (SELECT ToRepId, ToCtr FROM pred)
WITH isRoot AS (SELECT RepId, Ctr FROM set
                  WHERE (RepId, Ctr) NOT IN overwrites)
WITH isLeaf AS (SELECT RepId, Ctr FROM set
                  WHERE (RepId, Ctr) NOT IN overwritten)
WITH RECURSIVE isCausallyReady AS (
    SELECT * FROM isRoot
    UNION [ALL]
    SELECT pred.ToRepId, pred.ToCtr
    FROM pred, isCausallyReady
    WHERE pred.FromRepId = isCausallyReady.RepId
    AND pred.FromCtr = isCausallyReady.Ctr
)

SELECT set.key, set.value
FROM set, isCausallyReady, isLeaf
WHERE set.RepId = isCausallyReady.RepId
AND set.Ctr = isCausallyReady.Ctr
AND isCausallyReady.RepId = isLeaf.RepId
AND isCausallyReady.Ctr = isLeaf.Ctr
```

(b) The MVR key-value store including causal broadcast in SQL.

Figure 1.4: The MVR key-value store including causal broadcast in Datalog and SQL.

a causal broadcast expressed in Datalog. It additionally introduces the four predicates, “overwrites”, “isRoot”, “isLeaf”, and “isCausallyReady”, to only consider operations which are causally ready, to derive the state of the key-value store. The “overwrites” predicate is analogous to the “overwritten” predicate. The “isRoot” predicate captures the root nodes of the causal history, and the “isLeaf” predicate captures the leaf nodes of the causal history. Leaf nodes are the set of nodes that are not (yet) overwritten by other operations. The “isCausallyReady” predicate is defined recursively and captures the transitive closure of the `pred` relation, if starting from the root nodes of the causal history. I return to the computation of the transitive closure in Chapter 2 to explain the semantics of Datalog. Moreover, I use the key-value store CRDT Datalog query throughout this work as an ongoing example because of its interesting properties: It is a simple yet realistic example which makes use of recursion and negation, two features that I deem as important for expressing CRDTs.

While SQL has the advantage of being more widely known, Datalog’s syntax excels at concisely expressing recursion and composition [16]: Figure 1.4b shows an equivalent SQL query which I structure with the same subqueries as in the Datalog example, but the SQL query is more verbose and arguably less readable. Especially, I want to avoid the cumbersome syntax around recursive common table expressions which remains unpopular even within the SQL community [17]–[19]. The semantics of recursion in both Datalog and SQL are based on least fixed point iteration, thereby sharing the same limitations, such as requiring monotonicity of the computation to guarantee the fixed point’s existence and uniqueness. This precludes the use of negation in recursive queries, as I elaborate on in Section 2.1.2 when introducing stratified negation. Hirn et al. [18] discuss these limitations in more detail in the context of SQL and propose alternatives. Beyond this, SQL is based on multisets and Datalog on sets but Datalog can be extended to accommodate multisets as well, as I do when defining my Datalog dialect in Section 3.2. While SQL offers support for aggregates, Datalog requires an extension to support aggregates [20]. Yet, aggregates are not required to express the CRDTs of this work. Hence, there is no fundamental difference for my use case in terms of their semantics and expressiveness. Therefore, I prefer Datalog because of its syntax which allows for compact and composable queries. Composability is key to enabling reusability. Furthermore, there exists some recent research about defining Datalog over arbitrary semirings [16], [21], which may open up new avenues for Datalog’s expressiveness in the future.

The example also demonstrates why atomic writes to the database are important. Write w_2 updates both the `set` and `pred` relations. If the query reads state in which only the `set` relation, but not the `pred` relation, has been updated, the query incorrectly deems $set_4^{r_1}(k_1, v_5)$ as a new root and again returns the invalid result from above.

While the issue of this section can be ignored by assuming a causal broadcast either on the application or on the database layer, I think that queries benefit from having the full causal history available. It provides them with the ability to detect when operations are concurrent, and use that information to adjust their conflict handling to perform custom resolution logic.

1.3 Query Execution

CRDTs are often used in near-real-time collaboration settings which define the access patterns the query engine has to deal with. Near-real-time collaboration results in frequent updates both from the local and remote replicas and each update triggers a (re)evaluation of the CRDT query at each replica. Each individual update is usually small compared to the total size of the causal history. Only if a user has been working offline, they may send a larger batch of updates when they come back online. To aggravate the issue, operation-based CRDTs derive their state from a *monotonically growing* set of operations.

This is a poor fit for traditional query engines, which are stateless, meaning that they "forget" any work done for previous evaluations of the query, and start from scratch every time. With a monotonically growing set of operations, this implies a linear growth of the time it takes to evaluate the query, implying a threshold upon which the query engine is no longer able to evaluate the query within an acceptable time budget.

To address this issue, I focus on exploring incremental view maintenance (IVM) for this work. The promise of IVM is that it can incrementally maintain a view defined by a query, while only doing work relative to the size of the query's inputs' *change since the last evaluation*, as opposed to doing work relative to the size of the query's *full inputs*, as happening with traditional query engines. This promise may render query evaluations independent of the size (and correlatedly age) of the causal history but only dependent on the size of the accrued updates since the last evaluation.

2 Background

2.1 Datalog

Datalog [20] is a declarative logic programming language invented in the 1980s and is a subset of Prolog. It is primarily used for expressing queries to retrieve data in database systems. A Datalog program (or query) consists of a set of rules, which are used to derive new facts from existing ones. Unlike SQL, Datalog has not been attempted to be formally standardized, and several dialects of it exist. The explanations of this section focus on "conventional" Datalog and I define my own dialect in Section 3.2.

2.1.1 Syntax and Semantics

Rules define *predicates* which contain facts (also called tuples). Syntactically, rules are expressed in the form of Horn clauses, which are logical implications adhering to this structure:

$$\underbrace{\underbrace{r}_{\text{head}} \text{ :- } \underbrace{\underbrace{a_1}_{\text{atom}}, \ldots, \underbrace{a_n}_{\text{atom}}}_{\text{body}}}_{\text{rule}} . \tag{2.1}$$

The left-hand side of a rule is called *head*, the right-hand side *body*, and they are separated by a ":-" (colon-dash). A head consists of an identifier, which also defines the name of the predicate whose definition the rule contributes to, as well as a comma-separated list of expressions which may reference variables defined in the rule's body. A body is a comma-separated sequence of *atoms* followed by a trailing "." (dot). An atom either references another *predicate* (hereafter predicate atom) to bring some of its variables into scope or imposes a boolean condition (hereafter condition atom). A condition can either restrict a variable's value range, e.g. $x = 3$, or specify a relationship with another variable, e.g. $x = y$.

It is permitted to have multiple rules share the same head, that is, they have the same identifier and arity of expressions. In that case, they jointly define the predicate named after their heads' identifier. A rule (a predicate) is said to be *self-recursive* if it references itself in its (one of its) body (bodies). Similarly, several distinct rules (distinct predicates) are said to be *mutually recursive* if they reference each other in their bodies.

Semantically, a rule can be read from right to left: The body's atoms are connected with conjunctions and all variable assignments that satisfy that term form a new *fact* of the predicate which the rule defines. Here, Datalog's close relationship to first-order logic becomes apparent: Every rule is implicitly allquantified over its variables. If $x_1, \dots x_m$ are the variables of the rule's body, a rule can be read as "for all $x_1, \dots, x_m$ it holds that a_1 and … and a_n imply r". Mathematically, a rule can be expressed as[1]:

$$\forall x_1, \dots, x_m : a_1(x_1, \dots, x_m) \land \dots \land a_n(x_1, \dots, x_m) \Rightarrow r(x_1, \dots, x_m) \tag{2.2}$$

If there are multiple rules with the same head, their bodies' conjunctions are connected through a disjunction [16]. *Fact rules* are special rules without a body (with $n = 0$), e.g. r :- ., and define *extensional database predicates (EDBPs)*, whose facts are called *base facts*, are externally given, and assumed to be unconditionally true. Regular rules with a non-empty body (with $n > 0$) define *intensional database predicates (IDBPs)* and their facts are called *derived facts*. Facts are an ordered list of *fields* that can assume basic scalar types such as strings, numbers, or booleans. Due to their ordering, fields are accessed through positional indexing. Moreover, for a regular rule to be valid the *range-restriction property* must be satisfied [20]: It demands that every variable occurring in the head of a rule must also occur at least once in a predicate atom of its body, to avoid a dangling reference to a variable.

Allowing rules to be recursive equips Datalog with the ability to express repeated computations, therefore requiring a termination condition. Datalog uses least-fixed point semantics, under which the computation terminates if an additional iteration of applying a program's rules does contribute further derived facts to its result anymore *for the first time*. For a more formal treatment of Datalog's semantics, I refer to [20] which provides an overview on three equivalent formalisms to precisely define Datalog's semantics: Model-theoretic, fixpoint-theoretic, and proof-theoretic.

Due to the use of relational algebra as the basis for an intermediate representation (IR) in Chapter 3, I point out some similarities and differences to it, as well as to SQL, because it is the predominant frontend to relational algebra. The counterpart of a predicate in relational algebra is a *relation*, while the equivalent in SQL is a *table*. They all define a name under which facts (tuples in relational algebra; rows in SQL) are made available, whose fields contain scalar values. Datalog uses positional indexing to access fields, whereas in relational algebra and SQL, fields are accessed by their name. Furthermore, Datalog's predicates usually offer set semantics, and SQL's tables

[1]Not every atom or head must reference all variables.

are defined as *multisets* (bags) which permit duplicates. The semantics of relations in relational algebra depend on the context and can be either set-based or multiset-based.

```
edge(from, to, weight)        :- .
closure(from, to, weight, 1) :- edge(from, to, weight), from = 2.
closure(from, to, cweight + weight, hopcnt + 1)
                             :- closure(from, via, cweight, hopcnt),
                                edge(via, to, weight).
```

Figure 2.1: An exemplary computation of a graph's transitive closure with Datalog.

Figure 2.1 illustrates the concepts for the computation of a graph's transitive closure with Datalog. It contains three rules which define the two predicates, "edge" and "closure". Due its empty body, the "edge" rule defines an EDBP whose facts contain the fields "from", "to", and "weight" and define an edge in a directed, weighted graph. The first two fields specify *from which node to which other node* an edge points to, while the third field specifies *the weight of the edge*. Their types are not specified here but can be assumed to be (non-negative) integers. The facts of the "edge" predicate are not computed by Datalog but are assumed to be given externally, e.g., through an insertion into the database. The "closure" IDBP is defined through the last two rules sharing the same head. The last rule's body consists of two atoms: The first one references itself, rendering the rule self-recursive, and the second one references the "edge" predicate. Together they bring the variables "from", "via", "cweight", "hopcnt", "to", and "weight" into scope. The appearance of the same variable "via" in both atoms implies that their values must be equal, i.e., $a_1(x), a_2(x)$ is a shorthand for $a_1(x_1), a_2(x_2), x_1 = x_2$. Next to defining the name of the predicate, the head specifies that the "from" and "to" variables are exposed as the first two fields for facts from the "closure" predicate, and that its third and fourth field are defined through their respective expressions. The second rule's body contains the "edge" predicate atom and a condition atom which restricts the "from" variable to be equal to the value 2.

The "closure" IDBP defines the actual computation of the graph's transitive closure. The second rule specifies the computation's starting point, that is, all pairs of nodes which are reachable through a path of length one (direct edge), whose first entry is the node with id 2. The third rule takes all node pairs identified so far, which are reachable through paths of length n, and discovers new node pairs connected through a path of length $n + 1$, while keeping track of the cumulative weight of the path and the number of hops. The computation terminates if a (re)application of the rules on top of the currently derived facts does not produce new facts, upon which the least-fixed point is attained.

Least-fixed point computations can only solve problems whose solutions are monotonically growing, as otherwise the least-fixed point solution may not terminate at the optimal solution. This can be thought of being stuck at a local minimum instead of finding a global optimum in non-convex optimization. Another issue, not necessarily tied to least-fixed point computations but shared with all termination criteria, is that it is impossible to prevent non-terminating computations in general. Due to keeping track of the cumulated weight and the number of hops, the computation in Figure 2.1 is only guaranteed to terminate if the graph is cycle-free. With cycles, the computation would walk cycles endlessly and constantly discover higher cumulated weights, for node pairs which are part of a cycle. A discussion of alternatives to fixed points in the context of SQL can be found in [18].

2.1.2 Negation Extension

So far I presented *positive* Datalog, which does not allow negation of atoms. This leaves Datalog's expressiveness quite limited, as, for instance, it cannot express relational algebra's set difference operator or the examples from Figures 1.2a and 1.4a. However, introducing negation without any restrictions onto Datalog causes semantic issues, as Figure 2.2 demonstrates:

```
human(name)     :- .
isLiar(name)    :- human(name), not isHonest(name).
isHonest(name) :- human(name), not isLiar(name).
```

Figure 2.2: A Datalog program with negation but unclear semantics.

Assuming that the "human" EDBP only contains the fact $\{epimenides\}$[2], and trying to intuitively evaluate the "isLiar" IDBP, forces the evaluation to assess the "isHonest" IDBP for *epimenides*, which in turn triggers the evaluation of the "isLiar" IDBP for *epimenides* again, and so forth. Hence, carelessly introducing negation in Datalog causes undefined semantics in case of recursive rules which use negation as part of their recursive definition. To (quite literally) break this cycle, *semipositive* Datalog is introduced first, which then enables the definition of the semantics of *stratified* Datalog, both of which are syntactic restrictions on the use of negation and recursion [20]. While there are other approaches to reimpart unambiguous semantics to Datalog with negation, *stratified negation* is the predominant one [20].

[2]This is inspired by the liars paradox, whose first instance is (not quite correctly) attributed to the Cretan philosopher Epimenides.

Semipositive Datalog only allows the negation of EDBPs, but not IDBPs, as part of the body of a rule defining an IDBP. Furthermore, it demands that every variable occurring in the body of a rule must occur in at least one positive (non-negative) predicate atom, which is referred to as the *safety condition* [20]. If it was not satisfied, the result of a query would not only depend on the actual content of a database, and may be infinite. The semantics of negative EDBPs follows the intuition that the negation of an EDBP within the body of an IDBP (together with the safety condition) behaves similarly to the set difference operator in relational algebra.

Stratified Datalog relaxes the restriction a bit and allows the negation of IDBPs within a rule, as long as the predicates are *stratifiable*. A Datalog program is stratifiable if its predicates can be partitioned into *ordered* strata $P_1, \ldots, P_n$ such that:

- If the predicate p_b occurs positively in the body of a rule defining the predicate p_h, then $p_b \in P_i$, $p_h \in P_j$, and $i \leq j$.
- If the predicate p_b occurs negatively in the body of a rule defining the predicate p_h, then $p_b \in P_i$, $p_h \in P_j$, and $i < j$ [20].

To compute a stratification of a negative Datalog program P, its *precedence graph* G_P can be utilized [20]. It is a directed graph whose nodes are the predicates of P. Furthermore, there is

- a *positive* edge (unlabeled in Figure 2.3) from predicate p_i to predicate p_j if p_i occurs in the body of a rule defining p_j as well as
- a *negative* edge (labeled with $\neg$ in Figure 2.3) from p_i to p_j if p_i occurs *negatively* in the body of a rule defining p_j.

The precedence graph can be used not only to check whether a Datalog program is stratifiable but also to compute a stratification of it. The former is enabled by the theorem that a Datalog program is stratifiable if and only if its precedence graph G_P does not contain a cycle with a negative edge [20]. For the latter, two steps are required. First, the precedence graph's strongly connected components form the *strata* of the Datalog program. Second, to find a valid sequence of the strata, the strongly connected components have to be topologically sorted.

The resulting order of strata is what imparts the semantics to stratifiable Datalog programs: Every stratum P_{i+1} is evaluated after its previous stratum P_i as if it was a *semipositive* Datalog program with the IDBPs of P_{i+1}, which are defined in a lower stratum, treated as if they were EDBPs. The semantics are well-defined because next to their existence, which is given by construction, they are also unique: Any freedom

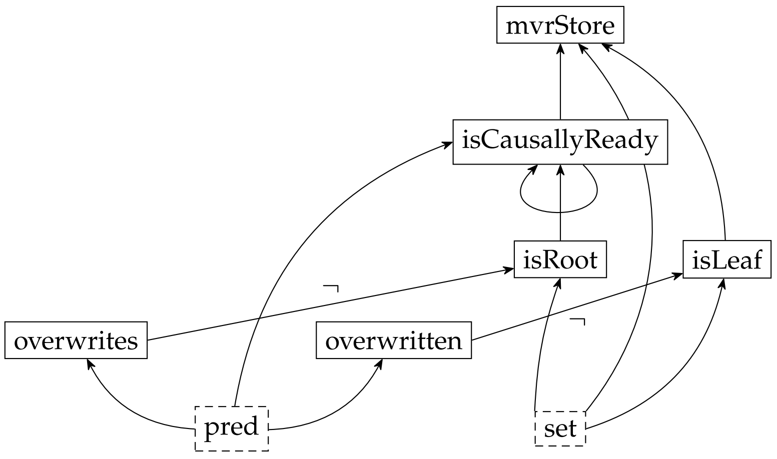


Figure 2.3: The precedence graph of the MVR key-value store CRDT of Figure 1.4a.

in choosing the strata and their order does not lead to different results, as long as the underlying data is the same [22].

Figure 2.3 shows the precedence graph of Figure 1.4a. It contains two negative edges but since they are not part of a cycle, the program is stratifiable. For example, one possible stratification is $P_0 = \{set, pred, overwrites, overwritten\}$ and $P_1 = \{isRoot, isLeaf, isCausallyReady, mvrStore\}$. The only cycle is the self-recursion of the "isCausallyReady" predicate, denoted by a positive self-loop edge. In general, if a precedence graph contains a cycle involving multiple nodes, the program contains *mutually recursive* predicates. The Datalog dialect I define in Section 3.2 sits in between semipositive and stratified Datalog: While it permits the negation of IDBPs in the body of a rule just like stratified Datalog, and it supports non-negated self-recursive rules, it does not support mutually recursive predicates.

2.2 CRDTs and Coordination-Free Environments

Some distributed applications are required to continue operating in the presence of network partitions because they want to support offline use cases (high availability). Additionally, they may want to support fast local writes because they cannot afford to wait for a network round trip to happen until the write becomes visible (low latency). For instance, collaborative editors may want both properties to allow users to work regardless of their network connectivity as well as to provide a responsive user experience, offering latency bounded by local disk access instead of a network round trip. The high availability and low latency properties define coordination-free environments and set them apart from distributed systems that require coordination. Yet, reads may be stale insofar that they do not always reflect the full global state. As a consequence, application specific invariants may be violated on the aggregate level after convergence, even though each write individually respected the invariants, based on their respective replica's state at write creation time. As long as applications can tolerate temporary violations of their invariants, they can take advantage of the high availability and low latency properties of coordination-free environments.

Convergent replicated data types (CRDTs)[3] are one solution for implementing coordination-free environments. They address the challenges of coordination-free environments by augmenting writes with additional metadata (1) to preserve the user's intention in the "best" possible manner and (2) to ensure the convergence of replicas, even after temporary divergence. CRDTs allow replicas to be offline for extended periods of time while still permitting them to write (and read) to (from) their local state at any time without having to coordinate with other replicas. When replicas come back online, they exchange their local state and converge to a common state, which is guaranteed to be the same for all replicas that have delivered the same set of updates.

The literature defines two classes of CRDTs: state-based and operation-based which both have different requirements for correctness. Let S be the set of all possible states of a CRDT. State-based CRDTs require a merge function $\sqcup : S \times S \rightarrow S$ which must be commutative, associative, and idempotent. Operation-based CRDTs require that the operation functions $op_i : S \rightarrow S$ are commutative and applied exactly once. Both models must adhere to these properties under the *strong eventual consistency* model which demands three properties [23]:

1. **Eventual Delivery**: All updates are eventually delivered to all replicas.
2. **Termination**: All method executions terminate.
3. **Convergence**: All replicas that have delivered the same set of updates are in an equivalent state.

Verifying the correctness of a CRDT is a complex, error-prone task [4], [5], even for people familiar with distributed systems, and currently has to be done for each CRDT individually. If, however, the set of operations performed on a CRDT *is* the state, the

[3]Although CRDTs are often referred to as *Conflict-free* Replicated Data Types, I prefer the term *convergent* here because the former term may be a bit misleading, as conflicts can still occur, e.g., in case of concurrent writes to the same "location" of a the data type. I think that CRDTs are better characterized by their property that diverging replicas eventually *converge* to the same state, given the delivery of the same set of updates. While that state may comprise conflicts, replicas uniformly agree upon them (and their order).

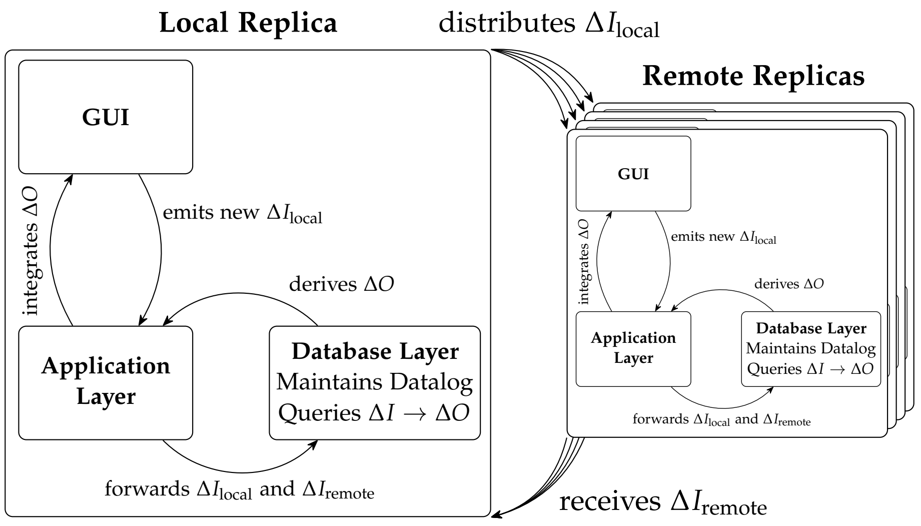

Local Replica
distributes ΔI_{local}
Remote Replicas
GUI
integrates ΔO
emits new ΔI_{local}
derives ΔO
Application Layer
Database Layer
Maintains Datalog
Queries $\Delta I \to \Delta O$
forwards ΔI_{local} and ΔI_{remote}
receives ΔI_{remote}

Figure 2.4: Overview of the system architecture from the perspective of a local replica.

merge function can be defined as the set union, for which the properties of commutativity, associativity, and idempotence hold. The convergence property demanded by the strong eventual consistency model is then also trivially satisfied because the state is by definition the set of all (delivered) operations. Moreover, applying any *pure*, i.e., deterministic and side-effect-free, function $f : S \to T$ on the state preserves the convergence property, as the function is applied on all replicas in the same deterministic way. T is an arbitrary set of all possible derived states f can map to. This approach to CRDTs is known in the literature as *pure operation-based replicated data types* [24], [25], and is used in practice in the Automerge CRDT [1]. This work explores defining the pure function f as a Datalog query over the state comprising the operations of a CRDT. As Datalog evaluation is deterministic, the convergence property remains satisfied.

Figure 2.4 provides an overview of a potential system architecture. The *database layer* maintains the views ΔO, as defined by the CRDT Datalog queries formulated on the *application layer*, in response to updates from both the local ΔI_{local} and remote replicas ΔI_{remote}. The application layer is responsible for forwarding updates to the database layer but forwarding could equally well happen on the database layer. As explained in Section 1.2, the database layer must be capable of atomically updating all relations of a write to safeguard the CRDT queries against reading inconsistent input state.

Although Figure 2.4 illustrates a peer to peer architecture, different network topologies are also possible. For instance, a star network topology with a central server could be used for more efficient update dissemination. However, the issue of update dissemination (and update integrity) is not the focus of this work and various approaches are discussed in the literature [26]–[29]. I only make the basic assumption of some network and protocol which ensures that each update will eventually be delivered to all replicas. Furthermore, replicas are assumed to be non-byzantine.

2.3 Incremental View Maintenance

Incremental view maintenance (IVM) [30] deals with the problem of maintaining an output view derived from a set of input relations in response to changes in these inputs. It computes the *changes* to the output view which accrued through changes in its inputs since the last evaluation. This renders the consumers of the output view *stateful*, unlike consumers of non-incremental queries which are stateless and obtain the full output view upon every evaluation. The goal of IVM is to maintain the view *efficiently*, that is, faster than recomputing the view from scratch upon an input change. In theory, this allows maintaining output views in near-real-time over large inputs which are subject to frequent changes, which would otherwise be prohibitively expensive to recompute from scratch every time. The promise of IVM is to do only work proportional to the size of the *input changes*, rather than the size of the *full inputs*, for a (re)evaluation of the output view. This draws a clear line to caching based approaches, e.g., periodically refreshed materialized views, whose evaluation is still proportional to the size of the full inputs. While they may provide low read latency, the output view may be stale. IVM tries to offer both low read latency and fresh data.

There are multiple approaches to IVM. Traditionally, IVM relies on algebraic transformations within the framework of relational algebra [30]–[32]. For a view V, its derivative ΔV has to be found to maintain it incrementally. Although this approach is not used in this work, I provide a small example for finding the derivative of a join. Let R and S be two relations and the view of interest is their equijoin $V := R \bowtie S$ on some field (not relevant here). Furthermore, ΔV, ΔR and ΔS are the set of changes to V, R and S, respectively, since their last evaluation. The state of V at the last evaluation is denoted as V_0 and the state of V which includes the changes is denoted as V_1. Then:

$$\begin{aligned} V_1 &= V_0 + \Delta V \\ \Leftrightarrow (R + \Delta R) \bowtie (S + \Delta S) &= R \bowtie S + \Delta V \\ \Leftrightarrow R \bowtie S + R \bowtie \Delta S + \Delta R \bowtie S + \Delta R \bowtie \Delta S &= R \bowtie S + \Delta V \\ \Leftrightarrow \Delta V &= R \bowtie \Delta S + \Delta R \bowtie S + \Delta R \bowtie \Delta S \end{aligned} \tag{2.3}$$

In the transformation from the first to the second line, the definitions of V_0 and V_1 are used. Then, the *bilinear* property (with respect to addition) of the join operator is used which allows the join to be distributed over the addition. A final transformation is applied to isolate ΔV and the well-known derivative of the join [33] is obtained, which can also be derived through the DBSP framework [11]. Unfortunately, the algebraic transformation approach is not applicable to all queries and struggles to handle more complex queries like recursive ones [11].

More recent approaches to IVM are *differential dataflow* [34] and the later *DBSP* [11], [35]. Their approach is fundamentally different from the algebraic one. At first, they leave relational algebra behind and instead define a general framework for expressing incremental computations over streams of insertions and deletions. Then, they show that the expressiveness of the framework is rich enough to represent relational algebra, including recursive queries, and by extension SQL and Datalog.

Although DBSP is more recent than differential dataflow, they make different tradeoffs instead of one superseding the other. The main advantage of DBSP is its modular theory: If a new operator can be expressed as a DBSP circuit, it is incrementalizable and DBSP's framework can be used to obtain an efficient implementation of it [11]. In contrast, there is no general recipe to express an arbitrary operator as a differential dataflow operator. This work chooses DBSP over differential dataflow for three reasons. First, its open-source implementation [36] provides a rich, (somewhat) documented API[4] to build upon. Second, it is said to be less complex while still providing all the necessary ingredients to to express CRDTs for my use case. Third, differential dataflow's advantage of being able to handle out-of-order input updates is not relevant for my approach to CRDTs, as it is based on sets and set union to integrate updates, which is a commutative operation already.

DBSP relies on the concept of $\mathbb{Z}$-sets [37][5]. With $\mathbb{Z}$-sets, each tuple of the $\mathbb{Z}$-set is tagged with an integer, called $\mathbb{Z}$-weight, which has a different meaning in different contexts. If the tuples represent data, the $\mathbb{Z}$-weight is non-negative and indicates the multiplicity of the tuple. A negative $\mathbb{Z}$-weight has no meaning. If the tuples represent changes to data, a positive $\mathbb{Z}$-weight indicates an insertion and a negative $\mathbb{Z}$-weight indicates a deletion of a tuple. Hence, $\mathbb{Z}$-sets allow representing both data and changes to data with the same construct.

Roughly speaking, DBSP calls a query plan a *circuit* and it can be assembled from the available operators on *streams*. Circuits can be nested to express recursive computations, in which case the outer circuit is called the *parent circuit* and the inner circuit is called the *child circuit*. Streams are an infinite sequence of elements from $\mathbb{Z}$-sets. Oper-

[4]While the documentation is not perfect and the API surface quite involved, it is sufficient to start building with DBSP. As part of this work, I have contributed a pull request to improve the documentation of DBSP's tutorial on recursive queries.

[5]The paper calls them $\mathbb{K}$-relations because it operates in a more general context of tagging tuples with elements from a commutative semiring $\mathbb{K}$. As DBSP uses the integers $\mathbb{Z}$ as an instance of the commutative semiring, they are named $\mathbb{Z}$-sets.

ators used in this work are *distinct*, *map* (for projections), *filter* (for selections), *join*, *plus* and *minus* (for multiset union and difference), *antijoin* (for set difference with respect to some key), as well as the *delta0* operator (required for importing streams from a parent circuit into a child circuit). Operators can be chained to form a circuit, which can be viewed as a directed acyclic graph of operators similar to operator trees in relational algebra. Its root nodes are the input streams and the leaf node represents the output stream, emitting the resulting tuples of the query. Each input stream provides a handle to feed in updates, which are processed according to the circuit after calling the *step* method on the root circuit. The output stream's handle can be used to obtain the result tuples.

There are *stateless* and *stateful* operators. For instance, every *linear* operator, such as a *filter* or a *map*, is stateless, meaning that it does not have to maintain any state between calls to the *step* method. On the other hand, bilinear operators, such as a *join*, are stateful and have to maintain state between calls to the *step* method. This has an important implication for query optimization [11]: Unlike with non-incremental query processing, the query plan is fixed and cannot adapt to changes in the inputs after the circuit has been assembled, *without* having to construct a new circuit according to the reoptimized query plan and then having to feed in all updates again to hydrate the stateful operators' state.

3 Implementation

This chapter outlines my solution to the problem motivated in Chapter 1. A fundamental prerequisite to assessing if CRDTs defined by queries executed on an incremental query engine is a viable approach, is to have such a query engine available. Unfortunately, the landscape of incremental Datalog query engines is sparse and I had to implement one myself. The engine is written in Rust and its source code is available on GitHub[1].

At the core of my query engine is a tree-walk interpreter that executes a query plan by delegating all relational computations to the DBSP library, while handling all operations on scalars, such as arithmetic and boolean expressions, itself. The query plan is represented in an intermediate representation (IR) which is a small programming language that supports variables, functions, and static scopes. Particularly, the IR supports relational operations, such as selections, projections, and joins, to represent a query plan. The IR and the interpreter are described in Section 3.1. Furthermore, the engine has a Datalog frontend. Due to Datalog's lack of standardization, I design a Datalog dialect which is discussed in Section 3.2. Finally, the translation from an abstract syntax tree (AST) of a Datalog program to a reasonably efficient query plan expressed in the IR is described in Section 3.3.

In general, query engines either interpret a query plan or compile the query plan into an executable program tailored to the specific query. I choose to implement a query engine based on interpretation for several reasons:

1. **Less complexity.** An interpreter is simpler to implement and to debug than a compiler. For an explorative project, this is a significant advantage.

2. **No compile time overhead.** Rust is known for its long compile times, potentially offsetting some possible performance gains of compiled query execution.

3. **Easier integration.** A library is easier to integrate into a larger system than a compiler. I think this is particularly relevant for applications that do not exclusively run on big servers but may be accommodated on smaller edge devices, which may not afford bundling a full compiler.

[1]`https://github.com/lstwn/masterthesis`

My motivation to use an IR based on relational algebra is twofold. The first one is query optimization possibilities. Although not being the focus of this work, query optimization on relational algebra has been studied for decades [38] and this approach opens the gate to leverage this research. Query optimization on Datalog has also been studied but to a lesser extent. Its results could also be applied as part of the Datalog frontend prior to the translation into the IR. Second, an IR provides a layer of abstraction in two ways. It allows multiple frontends to be implemented, such as a SQL frontend, as long as the frontend can be translated into the relational algebra IR. Moreover, it makes it easier to change the underlying incremental computation framework. If there is a way to implement the relational operators with differential dataflow for example, the IR can be executed with it. This presents an interesting future work direction, to better understand both approaches' differences and performance characteristics. Another possibility is to additionally offer a non-incremental processing mode and let the user choose which one to use.

3.1 Intermediate Representation in Relational Algebra

My language supports the following scalar types: string, integer, boolean, char, and null. Operations on scalars match what most of today's programming languages offer. There is support for arithmetic operations (`*, /, +, -`), logical operations (`and, or, not`), and comparisons (`>, >=, ==, <, <=`), as well as groupings through parenthesis in expressions. The IR's variables belong to a single static scope (also lexical scope). Static scopes are named after their property that it is *statically known*, i.e., without any program execution, to which exact variable a variable identifier points to at any given point in the program. Besides storing scalar values, variables can also store relations and functions. The latter renders functions first-class citizens and opens the door to code reuse and encapsulation, as functions store snapshots to their static environment in which they are defined. These functions are also known as closures.

To generalize over arbitrary relations, relations' tuples are represented as a sequence of scalars together with a schema for accessing its fields by name. The schema is represented as another sequence of the same length as of the tuple. Each entry contains the name of the field and if it is active or not. I refer to the schema of relation R as $sch(R)$. Figure 3.1 lists the supported relational operators of the IR. These are enough to express a wide range of queries and support my use case. To be able to execute queries correctly, the engine needs to keep track of the evolution of relations' schemas, as they are influenced by the operators of a query plan. Figure 3.2 shows the output schema for each operator from Figure 3.1.

The interpreter is invoked at two different stages of a query execution. First, it is

Operator	Notation	Description
Distinct	$distinct(R)$	Removes duplicate tuples from its input relation R.
Union	$R \cup S$	Merges its input relations R and S.
Difference	$R \setminus S$	Removes tuples from the left input relation R which are also present in the right input relation S.
Alias	$\rho_{name}(R)$	Renames its input relation R to *name* and allows referring to it (and its fields) by that name in subsequent operators.
Selection	$\sigma_{pred}(R)$	Filters tuples from its input relation R based on the predicate *pred*.
Projection	$\pi_{[(name,expr)]}(R)$	Produces a new relation with fields defined by the list of name-expression pairs. All expressions are evaluated in the context of a tuple from its input relation R.
Cartesian Product	$R \times S$	Combines two relations by pairing every tuple from the left input relation R with every tuple from the right input relation S.
Equijoin	$R \bowtie_{[(lexpr,rexpr)]} S$	Like the Cartesian product but it only emits a pair if all left-hand-side expressions of the list of pairs (*lexpr*; evaluated in the context of a tuple from R) evaluate to equal values as all right-hand-side expressions (*rexpr*; evaluated in the context of a tuple from S).
Antijoin	$R \triangleright_{[(lexpr,rexpr)]} S$	Returns all tuples from the left input relation R that do *not* match any tuple from the right input relation S based on the list of expression pairs (which is evaluated as for the equijoin).
Fixed point Iteration	$\mu(acc, step)$	Executes *step* starting from *acc* for as long as there are changes to its up-to-now computed output, i.e., it stops once the *least fixed point* is attained.

Figure 3.1: Relational operators of the IR.

$sch(distinct(R)) = sch(R)$
$sch(R \cup S) = sch(R)$ (or $= sch(S)$)
$sch(R \setminus S) = sch(R)$ (or $= sch(S)$)
$sch(\rho_{name}(R)) = sch(R)$
$sch(\sigma_{pred}(R)) = sch(R)$
$sch(\pi_{[(name,expr)]}(R)) = [name]$ (projection's list of field names)
$sch(R \times S) = sch(R) \circ sch(S)$ ($\circ$ denotes concatenation)
$sch(R \bowtie_{[(lexpr,rexpr)]} S) = sch(R) \circ sch(S)$
$sch(R \triangleright_{[(lexpr,rexpr)]} S) = sch(R)$
$sch(\mu(acc, step)) = sch(acc)$

Figure 3.2: The schema of the output of the operators of the IR.

invoked to construct a DBSP circuit, i.e., a query plan in Database terminology, from the input IR program, which can then be executed by DBSP's runtime. I refer to this invocation as *query build-time*. Second, while DBSP executes the circuit, the interpreter is invoked to evaluate expressions and to access the values of variables. For instance, this happens whenever a selection's predicate or the expressions of a projection are evaluated. I refer to this as *query execution-time*.

Figure 3.3 shows the same computation of a graph's transitive closure from Section 2.1.1 but expressed in pseudocode which resembles the real representation of the IR but with some Rust-specific boilerplate omitted. On the top level, it shows the main statements of the IR program which are all variable assignments in this case. The defined variables, "edges", "base", and "closure", can then be referenced in subsequent statements, respectively. The "edges" variable is referenced by both the "ProjectionExpr" and the "FixedPointIterExpr". The "base" variable is referenced by the "FixedPointIterExpr". Due to statements returning their value as well (as long as they produce any), the "closure" variable's value is also the output of the IR program, resembling implicit returns known from Rust or Ruby. The "FixedPointIterExpr" requires all relations it wants to use to be specified within its "step" body because DBSP requires relations from parent circuits to be imported into child circuits via its *delta0* operator. To provide this, the "imports" and "accumulator" fields are used to specify the relations which have to be injected into the context of the fixed point iteration's body. Note that the "EquiJoinExpr" supports including a projection step, making a trailing "ProjectionExpr" of the fixed point iteration's body obsolete. The highlighted code in blue shows the code which is executed at query execution-time in the context of a tuple. The other code is executed at query build-time to construct the DBSP circuit.

```
edges   <- LiteralExpr(Relation { schema: ["from", "to", "weight"] })
base    <- ProjectionExpr {
             relation: VarExpr("edges"),
             attributes: [
               ("from", VarExpr("from")),
               ("to", VarExpr("to")),
               ("cweight", VarExpr("weight")),
               ("hopcnt", LiteralExpr(UInt(1))),
             ]
           }
closure <- FixedPointIterExpr {
             imports: ["edges"],
             accumulator: ("acc", VarExpr("base")),
             step: BlockStmt {
               stmts: [
                 ExprStmt {
                   expr: EquiJoinExpr {
                     left: AliasExpr { relation: VarExpr("acc"), alias: "cur" },
                     right: AliasExpr { relation: VarExpr("edges"), alias: "next" },
                     on: [(VarExpr("cur.to"), VarExpr("next.from"))],
                     attributes: [
                       ("from", VarExpr("cur.from")),
                       ("to", VarExpr("next.to")),
                       (
                         "cweight",
                         BinaryExpr
                           op: Operator::Add,
                           left: VarExpr("cur.cweight"),
                           right: VarExpr("next.weight")
                         }
                       ),
                       (
                         "hopcnt",
                         BinaryExpr {
                           op: Operator::Add,
                           left: VarExpr("cur.hopcnt"),
                           right: LiteralExpr(UInt(1))
                         }
                       )
                     ]
           }}]}}
```

Figure 3.3: The computation of the transitive closure from Figure 2.1 translated into the IR.

Prior to the execution, an IR program first undergoes a static pass to resolve all references to variables and functions except for tuples' variables. Tuples' variables are not yet available at query build-time but are injected later during query execution-time. Hence, they require dynamic resolution. Yet, resolving regular variables and functions statically has the advantage that the interpreter can avoid dynamic lookups, which require walking the program's static scope tree, at both query build and execution-time. Moreover, the static variable resolver pass fixes a problem with closures capturing their environment, which exists if environments are implemented with mutable instead of immutable data structures [39].

The ability to mark fields as active or inactive allows optimizing projections: If a projection only wants to pick a subset of a relation's fields and does not contain any expressions which require evaluations, the projection can be executed by just changing the relation's schema without having to run the interpreter on the relation's tuples during query execution-time.

3.2 Datalog Frontend

Figure 3.4 shows the grammar of my Datalog dialect[2], along with the syntax for literals as well as for a small expression language which is required to specify conditions on variables. The dialect follows "typical" Datalog syntax and semantics except for some modifications.

Traditionally, Datalog uses positional indexing to access variables from a predicate. I decided to use name-based indexing for two reasons: First, it aligns better with the IR which uses relational algebra that uses names to refer to relations' variables. Second, positional indexing is inconvenient in practice because predicates (or relations) with many columns occur in real-world database schemas, rendering positional indexing cumbersome to use. Moreover, variables starting with an underscore are ignored but can be used to make things more explicit.

The grammar also allows the "distinct" keyword to be prepended to the name of a rule's head. This causes the facts of the rule to be distinct, i.e., it removes duplicates from the rule's output. If the distinct keyword is not given, the resulting relation is a multiset, like in SQL. As Datalog uses set semantics traditionally, an explicit distinct operator is usually not necessary because it is implicitly given. Nevertheless, I deviate to allow for multiset semantics for two reasons: First, in practice it may be useful to be able to work with duplicates. Second, enforcing set semantics can be costly performance-wise, as it requires maintaining an index to check for duplicates. This

[2]The grammar does not include comment support for brevity. The implementation supports //-EOL-style comments between rules and between atoms of a rule's body.

```
// Core Datalog grammar.
program     = rule* EOF ;
rule        = head ":-" body "." ;
head        = "distinct"? IDENTIFIER "(" field ( "," field )* ")" ;
field       = IDENTIFIER ( "=" safe_expr )? ;
body        = ( atom ( "," atom )* )? ;
atom        = ( "not"? predicate ) | safe_expr ;
predicate   = IDENTIFIER "(" variable ( "," variable )* ")" ;
variable    = IDENTIFIER ( "=" IDENTIFIER )? ;

// Scalar expressions grammar.
safe_expr   = "(" expr ")" | comparison ;
expr        = logical_or ;
logical_or  = logical_and ( ";" logical_and )* ;
logical_and = comparison ( "," comparison )* ;
comparison  = term ( ( "==" | "!=" | ">" | ">=" | "<" | "<=" ) term )? ;
term        = factor ( ( "+" | "-" ) factor )* ;
factor      = unary ( ( "*" | "/" ) unary )* ;
unary       = ( "-" | "!" ) unary | primary ;
primary     = literal | IDENTIFIER | "(" expr ")" ;
literal     = BOOL | UINT | IINT | STRING | NULL ;

// Primitives and literals.
BOOL        = "true" | "false" ;
UINT        = DIGIT+ ;
IINT        = ( "-" | "+" )? DIGIT+ ;
STRING      = "\""<any char except "\"">*"\"" ;
IDENTIFIER  = ALPHA ( ALPHA | DIGIT )* ;
ALPHA       = "a".."z" | "A".."Z" | "_" ;
DIGIT       = "0".."9" ;
NULL        = "null" ;
```

Figure 3.4: The grammar of my Datalog Variant.

is similar to the situation with relational algebra and SQL. In most formal settings, relational algebra uses set semantics but nearly every SQL implementation in practice uses multiset semantics.

Besides that, new fields can be defined through expressions in the list of fields of a rule's head. Yet, due to name-based indexing, a name must be provided. The expressions can reference all variables which are in scope of the rule's body. My dialect prohibits mutual recursion and only permits self-recursion, i.e., any program's precedence graph must be acyclic except for self-loops. This is stricter than stratified Datalog which allows cycles, provided that these cycles do not contain a negative edge. This is a deliberate design choice to keep the implementation complexity manageable while still being sufficient for my CRDT use case. Section 4.1 shows the familiar key-value stores from Chapter 1 as well as a list CRDT in my Datalog dialect.

The parser itself is implemented with the help of the "nom" library [40] which is a parser combinator framework for Rust. Parser combinators are higher-order functions used to compose a parser from smaller parsers, which makes writing a parser for an existing grammar relatively straightforward, as long as unit tests are written for the input parsers first. Otherwise, the origin of a parsing bug may be hard to identify.

3.3 Translating Datalog to Relational Algebra

Due its declarative nature, Datalog does not specify how to execute a query and, in particular, leaves the problem of finding a valid execution order to the query engine. Therefore, a precedence graph is constructed as described in Section 2.1.2 from the *aggregated* rules of a Datalog program. Aggregated means that all rules with the same head are combined into an aggregated rule which contains all bodies of the original rules. This has the advantage that every predicate is now represented by a single aggregated rule. Then, a topological sort is computed on top of the precedence graph using Kahn's algorithm [41]. In addition to yielding a valid execution order that respects which predicate has to be computed before which other predicate, the algorithm also aborts in case of a cycle in the precedence graph, thereby detecting programs that are invalid due to use of mutual recursion. To avoid a false positive in case of a self-recursive predicate, the construction of the precedence graph omits self-loops.

Having found a valid execution order of the predicates, the aggregated rules are translated into the relational algebra IR. The topologically sorted predicates are mapped into a sequence of statements which assign the output of an aggregated rule to a variable with the same name. Then, subsequent rules which depend on the output of a previous rule can refer to the variable of a previous rule if necessary. The last predicate in the topological sort order implicitly becomes the output of the IR program because the IR supports implicit returns as well as statements (potentially) producing values. Due to the nature of topological sorting, the last predicate is the predicate with the most dependencies and implicitly assumed to be the *main predicate of interest* of the Datalog program. In case of a tie between multiple predicates, the discovery order of Kahn's algorithm is used to break the tie. This is a limitation in the implementation, as it does not support multiple main predicates. Yet, these may be useful in contexts where computations share dependencies for performance or code maintainability reasons, or where the results of multiple predicates are needed at the same time.

3.3.1 Translation of Aggregated Rules to the IR

This leaves the question of how to translate an individual aggregated rule (predicate) into its IR representation. The representation is similar to an operator tree in relational algebra. To illustrate the translation, I first discuss a *naive* translation which emerges from Datalog's reliance on first-order logic which can be translated to set-theoretic operations. Then, I introduce a more efficient translation that I have implemented for the query engine, called *subpar* translation. At first, the discussion precludes negative atoms and is limited to non-recursive predicates, both of which I introduce successively.

Rules with only positive atoms. Non-recursive rules without negative atoms can be naively translated into a Cartesian product of all predicates in the rule's body followed by a selection. To handle potential name collisions among the fields, each positive predicate atom is aliased with a unique prefix. Moreover, each referenced predicate is projected to only include the listed fields and to potentially rename them. The selection's predicate consists of *conjunctions* encompassing all comparison atoms in the rule's body as well as equality comparisons. For each variable occurring in $n > 1$ relations, $n - 1$ equality comparisons are included in the selection's predicate. Each equality comparison connects a pair of relations such that there exists a path connecting all n relations. Each comparison has the name of the variable on the both the left-hand and right-hand side of the comparison but prefixed with the relations' aliases, respectively. Finally, the list of expressions of the rule's head are projected from the selection's output.

Figure 3.5b shows the naive query plan for the "mvrStore" predicate (depicted in Figure 3.5a) of the key-value store CRDT from Figure 1.4a. After combining the three positive atoms, "set", "isCausallyReady", and "isLeaf", through a Cartesian product, the selection filters out all tuples that do not match with the variable reuse stated in the rule's body. As the last step, the projection confines the selection's output to the "Key" and "Value" fields given by the rule's head.

The *subpar* query translation improves upon the naive translation by avoiding the

Cartesian product where possible and instead folding the positive atoms into an operator tree of equijoins. This avoids large intermediate results of the Cartesian products through exploiting the selectivity of the join conditions. Additionally, the subpar translation eliminates unnecessary projections in two cases: If the input relations are not renamed, their projections can be omitted. Furthermore, any projection that is immediately preceded by a join can be coalesced into the join operator, as the join operator of the IR allows specifying a projection to apply onto its emitted tuples. The engine does not, however, optimize join ordering or apply other optimization techniques, such as predicate pushdown or expression simplification, which is why I call it *subpar*. These optimizations could be added in future work. Currently, the join order reflects the order in which the predicates appear in the rule's body.

Figure 3.5c shows the subpar query plan for the "mvrStore" predicate. It eliminates all Cartesian products and replaces them with equijoins. The projections and aliases can also be omitted, reducing the operator count from ten to two, while also avoiding Cartesian products' large intermediate results.

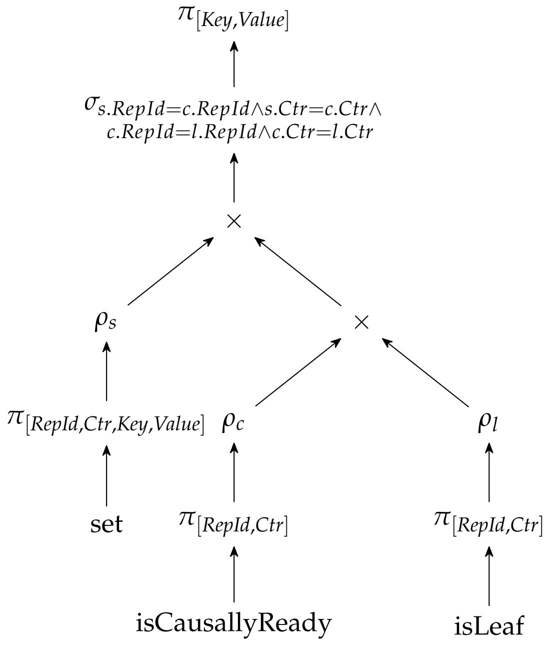
Rules with positive and negative atoms. To handle rules with both positive and negative atoms, all atoms in the rule's body are partitioned into positive and negative atoms. First, the positive atoms are translated into an operator tree as described above. Then, the negative atoms are folded into an operator tree of antijoins, starting from the positive atoms' operator tree. The partitioning ensures that the positive atoms are evaluated first. Due to the *safety condition* of negative Datalog (Section 2.1.2), it is guaranteed that all variables referenced in the negative atoms occur at least once in the positive atoms, too, rendering the antijoins well-defined. For the negative atoms, the antijoin operator is useful, as it does not require schema equality between its input relations, unlike the set difference operator. In theory, the set difference operator could also be used but it may result in projections cutting off fields to ensure schema equality between its inputs. Yet, in some cases, the previously cut off fields have to be joined back in later. To avoid this, I use the antijoin operator, which allows a set difference to be specified with respect to a list of expressions.

Figure 3.6b shows a query plan for a variant of the "mvrStore" predicate which inlines the "isLeaf" predicate (see Figure 3.6a). This causes the "mvrStore" rule to contain the negative "overwritten" predicate atom in its body. The colored rectangles highlight the partitioning of the atoms into positive and negative parts with their respective translations. The query plan for the positive atoms is the input to the antijoin operation to cover the negative "overwritten" atom. The final projection to the "Key" and "Value" fields remains the same as before. Yet, unlike the equijoin, the antijoin does not support including a projection step.

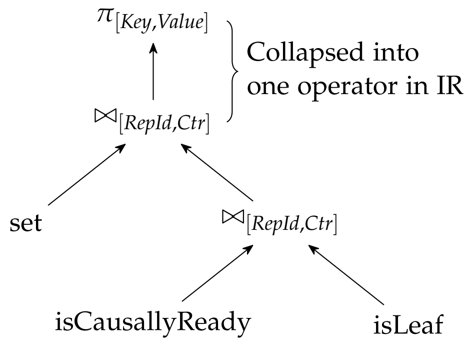
In case of multiple rules with the same head, every rule's translated operator tree is folded into a union. The use of the union is unproblematic here, as the equal heads of

```
mvrStore(Key, Value) :- set(RepId, Ctr, Key, Value),
                        isCausallyReady(RepId, Ctr),
                        isLeaf(RepId, Ctr).
```

(a) The "mvrStore" rule to optimize.



(b) The naive query plan.



(c) The subpar query plan.

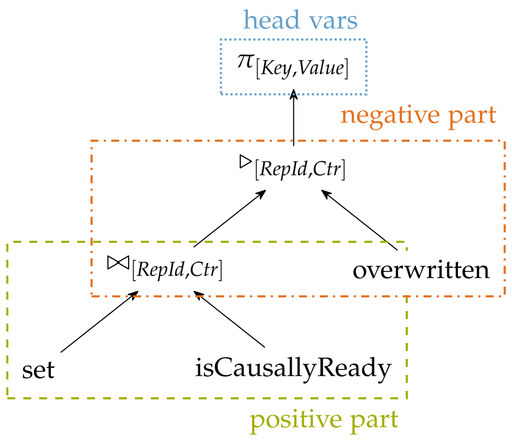
Figure 3.5: Different query plans for the "mvrStore" predicate of Figure 1.4a.

the rules ensure that the outputs of the operator trees share the same schema. If a rule's head is marked as "distinct", its output (after the projection to the heads' variables) is wrapped in a distinct operator to eliminate duplicates. Should *all* rules that share the same head be marked as "distinct", the union is performed first and only a single distinct is applied to the union's output, as opposed to applying a distinct operator to each rule's output and then performing the union, which would not only be less efficient but also semantically incorrect: If multiple rules produce the same output tuple, the union operator would not eliminate duplicates, as it is not a set operator but a multiset operator. Moreover, each rule with an empty body (defining an EDBP) is translated into a relation literal of the IR, specifying the relation's name and its field names which can be accessed.

Self-recursive rules. Self-recursive predicates are translated into a fixed point iteration of the IR, as shown in Figure 3.3. To do so, the fixed point iteration's "imports", "accumulator", and "step" (iteration body) have to be defined. They require partitioning the aggregated rule's bodies into *non-recursive* and *recursive* bodies, which are individually translated into operator trees as described above. The accumulator is the initial value of the fixed point iteration and given by the union of the *non-recursive* bodies' operator trees of the aggregated rule. The imports are all predicates that are referenced in the rule's *recursive* bodies, except for the recursive predicate itself, which does not require importing. The iteration body is the union of the *recursive* bodies' operator trees of the aggregated rule. In case any of the unions only contains a single input, the union is omitted and the sole input is used instead.

```
mvrStore(Key, Value) :- set(RepId, Ctr, Key, Value),
                        isCausallyReady(RepId, Ctr),
                        not overwritten(RepId, Ctr).
```

(a) The “mvrStore” rule with the “isLeaf” predicate inlined.



(b) Query plan with an antijoin to handle the negative “overwritten” atom.

Figure 3.6: A query plan for the modified “mvrStore” predicate of Figure 1.4a.

4 Evaluation

This chapter assesses the suitability of my Datalog dialect from Section 3.2 to express CRDTs-as-queries in Section 4.1, by implementing two classes of CRDTs in it. To evaluate the approach's viability in practice, Section 4.2 presents benchmarks of the implemented CRDTs which are executed on my query engine from Chapter 3.

4.1 CRDTs as Queries

Section 4.1.1 shows the familiar MVR key-value stores from Chapter 1 in my Datalog dialect. Section 4.1.2 implements a list CRDT in my Datalog dialect, which is adapted from [8].

4.1.1 Key-Value Stores

Figure 4.1 shows the MVR key-value store from Figure 1.2a (which assumes causal broadcast) in my Datalog dialect. Examples in this chapter explicitly state the EDBPs to communicate the predicates' schema to the query engine.

```
// EDBPs:
pred(FromRepId, FromCtr, ToRepId, ToCtr) :- .
set(RepId, Ctr, Key, Value)              :- .

// IDBPs:
distinct overwritten(RepId, Ctr)
                    :- pred(RepId = FromRepId, Ctr = FromCtr).
mvrStore(Key, Value)
                    :- set(RepId, Ctr, Key, Value),
                       not overwritten(RepId, Ctr).
```

Figure 4.1: The MVR key-value store without causal broadcast in my Datalog dialect.

Figure 4.2 provides a definition of the MVR key-value store from Figure 1.4a in my Datalog dialect. The query differentiates itself from the previous example by including the causal broadcast mechanism.

```
// EDBPs are omitted because they are shared with the previous example. IDBPs:
distinct overwritten(RepId, Ctr)
                    :- pred(RepId = FromRepId, Ctr = FromCtr).
distinct overwrites(RepId, Ctr)
                    :- pred(RepId = ToRepId, Ctr = ToCtr).
isRoot(RepId, Ctr)  :- set(RepId, Ctr, _Key, _Value),
                       not overwrites(RepId, Ctr).
isLeaf(RepId, Ctr)  :- set(RepId, Ctr, _Key, _Value),
                       not overwritten(RepId, Ctr).
isCausallyReady(RepId, Ctr)
                    :- isRoot(RepId, Ctr).
isCausallyReady(RepId, Ctr)
                    :- isCausallyReady(FromRepId = RepId, FromCtr = Ctr),
                       pred(FromRepId, FromCtr, RepId = ToRepId, Ctr = ToCtr).
mvrStore(Key, Value)
                    :- isLeaf(RepId, Ctr),
                       isCausallyReady(RepId, Ctr),
                       set(RepId, Ctr, Key, Value).
```

Figure 4.2: The MVR key-value store including causal broadcast in my Datalog dialect.

Both examples are similar to their counterparts from Chapter 1 expressed in "conventional" Datalog. Key differences are the inclusion of EDBPs, the use of name-based indexing, and the explicit use of the "distinct" operator to ensure that a predicate has set (instead of multiset) semantics.

4.1.2 List CRDT

List CRDTs enable replicas to converge to the same sequence of elements. Fundamentally, they support two kinds of operations: The insertion and the deletion of an element at a position. They are often discussed in the context of collaborative text editing in which the list maintains the sequence of characters as they appear in the shared text. Yet, most list CRDTs can go beyond this application and store arbitrary elements. Among others, Treedoc [42], Logoot [43], replicated growable arrays (RGA) [44], and Fugue [7] have been proposed. Causal trees [45] and timestamped insertion trees [46] are tree-based reformulations of RGA.

List CRDTs are more complex than key-value stores because they have to converge to the *same order* of elements across replicas under concurrent updates. The list CRDT presented here resembles the causal tree RGA formulation, which is based on the idea that insertions do not specify an index for their position, but instead each list element is assigned a unique, immutable identifier, and each insertion references the identifier of the element after which it wants to be inserted. This implies that deletions cannot fully remove an element, but have to leave *tombstones* behind, to avoid dangling references in case of an insertion after element x and a concurrent deletion of x. In case of concurrent insertions after the same element x, the inserted elements are ordered according to the total order induced by the identifiers. The identifiers can again be replica id and counter pairs, and they provide a total order by first comparing the counters and, in case of ties, by breaking them through comparing the replica ids. I initially focus only on insertions into the list, and discuss deletions later.

Logically, each insertion of an element into the list can be represented in a tree structure, where each node's *parent* is the element after which it is supposed to be inserted (but may not in the final list order due to concurrent insertions). The root element is some sentinel element that is always present and does not contribute to the list's content. A node's *children* are *descendingly* ordered by their identifiers. The number of children corresponds to the number of concurrent insertions after the parent element, i.e., siblings are a result of concurrency. The final list order is given by a depth-first, pre-order traversal of the tree. At its core, the Datalog query has to implement this depth-first, pre-order traversal of the tree.

Figure 4.3 shows the definition of a list CRDT in my Datalog dialect. To explain the query, I use Figure 4.4 as an example in which three replicas, with id 1, 2, and 3, insert elements into the list. Figure 4.4 visualizes the "insert(RepId, Ctr, ParentRepId, ParentCtr, Value)" EDBP with entries:

$$\{(2,1,0,0,'H'),(2,3,2,1,'E'),(1,3,2,1,'L'),(3,2,2,1,'L'),(1,1,0,0,'O'),(2,2,1,1,'!')\}$$

Every node (except for the sentinel root) depicts a (*RepId*, *Ctr*) pair and its black edge points to its parent defined by a (*ParentRepId*, *ParentCtr*) pair. The edge's label shows the value of the inserted element. The result of the depth-first, pre-order traversal is:

$$[(0,0),(2,1),(2,3),(1,3),(3,2),(1,1),(2,2)]$$

Despite the absence of a built-in list type in Datalog, I reproduce this result as an (unsorted) linked list of nodes with the help of the "nextElem" IDBP. Its definition is divided into three parts. First, the "firstChild" IDBP (Figure 4.3a) finds the first child of each parent (the dotted, green edges in Figure 4.4). To do so, it relies on the "laterChild" IDBP to find all parents with their children such that the children are not their first but a later child. Second, the "nextSibling" IDBP (Figure 4.3b) provides the next sibling of each child (the dashed, orange edges in Figure 4.4). Its definition makes use of the "laterSibling" and "laterIndirectSibling" IDBPs. The former finds

```
// Black edges in Figure 4.4.
insert(RepId, Ctr, ParentRepId, ParentCtr, Value) :- .
remove(ElemId, ElemCtr) :- .

laterChild(ParentRepId, ParentCtr, ChildRepId, ChildCtr) :-
    insert(SiblingRepId = RepId, SiblingCtr = Ctr, ParentRepId, ParentCtr),
    insert(ChildRepId = RepId, ChildCtr = Ctr, ParentRepId, ParentCtr),
    (SiblingCtr > ChildCtr; (SiblingCtr == ChildCtr, SiblingRepId > ChildRepId)).

// Dotted, green edges in Figure 4.4.
firstChild(ParentRepId, ParentCtr, ChildRepId, ChildCtr) :-
    insert(ChildRepId = RepId, ChildCtr = Ctr, ParentRepId, ParentCtr),
    not laterChild(ParentRepId, ParentCtr, ChildRepId, ChildCtr).
```

(a) Part 1: Definition of "insert" EDBP and "firstChild" IDBP.

```
sibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr) :-
    insert(Child1RepId = RepId, Child1Ctr = Ctr, ParentRepId, ParentCtr),
    insert(Child2RepId = RepId, Child2Ctr = Ctr, ParentRepId, ParentCtr).

laterSibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr) :-
    sibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr),
    (Child1Ctr > Child2Ctr; (Child1Ctr == Child2Ctr, Child1RepId > Child2RepId)).

laterIndirectSibling(Child1RepId, Child1Ctr, Child3RepId, Child3Ctr) :-
    sibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr),
    sibling(Child1RepId, Child1Ctr,
        Child3RepId = Child2RepId, Child3Ctr = Child2Ctr),
    (Child1Ctr > Child2Ctr; (Child1Ctr == Child2Ctr, Child1RepId > Child2RepId)),
    (Child2Ctr > Child3Ctr; (Child2Ctr == Child3Ctr, Child2RepId > Child3RepId)).

// Dashed, orange edges in Figure 4.4.
nextSibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr) :-
    laterSibling(Child1RepId, Child1Ctr, Child2RepId, Child2Ctr),
    not laterIndirectSibling(Child1RepId, Child1Ctr,
        Child2RepId = Child3RepId, Child2Ctr = Child3Ctr).
```

(b) Part 2: Definition of "nextSibling" IDBP.

```
distinct hasNextSibling(ChildRepId, ChildCtr) :-
    nextSibling(ChildRepId = Child1RepId, ChildCtr = Child1Ctr).

// Dashdotted, blue edges in Figure 4.4.
nextSiblingAnc(ChildRepId, ChildCtr, AncRepId, AncCtr) :-
    nextSibling(ChildRepId = Child1RepId, ChildCtr = Child1Ctr,
        AncRepId = Child2RepId, AncCtr = Child2Ctr).
nextSiblingAnc(ChildRepId, ChildCtr, AncRepId, AncCtr) :-
    insert(ChildRepId = RepId, ChildCtr = Ctr, ParentRepId, ParentCtr),
    not hasNextSibling(ChildRepId, ChildCtr),
    nextSiblingAnc(ParentRepId = ChildRepId, ParentCtr = ChildCtr,
        AncRepId, AncCtr).

distinct hasChild(ParentRepId, ParentCtr) :-
    insert(ParentRepId, ParentCtr).

nextElem(PrevRepId, PrevCtr, NextRepId, NextCtr) :-
    firstChild(PrevRepId = ParentRepId, PrevCtr = ParentCtr,
        NextRepId = ChildRepId, NextCtr = ChildCtr).
nextElem(PrevRepId, PrevCtr, NextRepId, NextCtr) :-
    not hasChild(PrevRepId = ParentRepId, PrevCtr = ParentCtr),
    nextSiblingAnc(PrevRepId = ChildRepId, PrevCtr = ChildCtr,
        NextRepId = AncRepId, NextCtr = AncCtr).
```

(c) Part 3: Definition of "nextSiblingAnc" and "nextElem" IDBPs.

```
distinct hasValue(ElemId, ElemCtr) :-
    // Fix for not being able to assign anything to the sentinel element.
    insert(ElemId = ParentRepId, ElemCtr = ParentCtr),
    ElemId == 0, ElemCtr == 0.
distinct hasValue(ElemId, ElemCtr) :-
    insert(ElemId = RepId, ElemCtr = Ctr),
    not remove(ElemId, ElemCtr).

nextElemSkipTombstones(PrevRepId, PrevCtr, NextRepId, NextCtr) :-
    nextElem(PrevRepId, PrevCtr, NextRepId, NextCtr).
nextElemSkipTombstones(PrevRepId, PrevCtr, NextRepId, NextCtr) :-
    nextElem(PrevRepId, PrevCtr, ViaRepId = NextRepId, ViaCtr = NextCtr),
    not hasValue(ViaRepId = ElemId, ViaCtr = ElemCtr),
    nextElemSkipTombstones(ViaRepId = PrevRepId, ViaCtr = PrevCtr,
        NextRepId, NextCtr).

nextVisible(PrevRepId, PrevCtr, NextRepId, NextCtr) :-
    hasValue(PrevRepId = ElemId, PrevCtr = ElemCtr),
    nextElemSkipTombstones(PrevRepId, PrevCtr, NextRepId, NextCtr),
    hasValue(NextRepId = ElemId, NextCtr = ElemCtr).

listElem(PrevRepId, PrevCtr, Value, NextRepId, NextCtr) :-
    nextVisible(PrevRepId, PrevCtr, NextRepId, NextCtr),
    insert(NextRepId = RepId, NextCtr = Ctr, Value).
```

(d) Part 4: Definition of “listElem” IDBP.

Figure 4.3: A list CRDT in my Datalog dialect, adapted from [8].

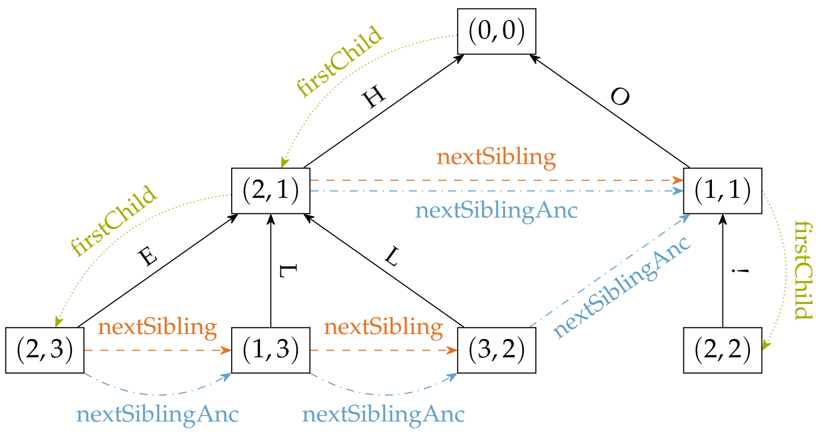


Figure 4.4: An example tree spanned by insertions into the list CRDT.

all later siblings of a child, and the latter finds all later siblings of a child that are not direct siblings but have at least one sibling in between them. Then, the next sibling of a child is given by filtering out all later *indirect* siblings from the later siblings. Third, the "nextSiblingAnc" IDBP (Figure 4.3c) gives the next sibling of each child, or, recursively, the next sibling of the parent if a child has no next sibling (the dashdotted, blue edges in Figure 4.4). Finally, the "nextElem" IDBP defines the depth-first, pre-order traversal by either yielding a link from the parent to its first child (as "previous" and "next", respectively) or a link defined by the "nextSiblingAnc" IDBP if the "previous" node has no child. This outputs the above result as a linked list of nodes where each node is defined by a (*RepId*, *Ctr*) pair:

$$\{(0,0,2,1),(2,1,2,3),(2,3,1,3),(1,3,3,2),(3,2,1,1),(1,1,2,2)\}$$

Every deletion generates an entry in the "remove" EDBP with the (*RepId*, *Ctr*) pair of the node to be deleted. Therefore, some nodes may not have a value assigned anymore, in which case they represent a tombstone and must be skipped in the output. To achieve that, the "listElem" IDBP (Figure 4.3d) is used, which relies on the "nextElem-SkipTombstones" and "nextVisible" IDBPs. The former is based on the "nextElem" IDBP but additionally contains longer paths derived from the "nextElem" predicate which skip tombstoned nodes. The "hasValue" IDBP checks whether a node has been tombstoned or not, and the first rule of its definition ensures that the sentinel root element (with id $(0,0)$) is always considered to have a value assigned to it. This is required to prevent the "nextVisible" IDBP from skipping the first list element. The "nextVisible" IDBP ensures that each node from the node pairs reported by the "next-ElemSkipTombstones" IDBP has a value assigned to it. Finally, the "nextElem" IDBP joins the non-tombstoned list elements' values to the pairs of nodes from the "next-Visible" IDBP. The final list result for the example from Figure 4.4 is:

$$\{(0,0,'H',2,1),(2,1,'E',2,3),(2,3,'L',1,3),(1,3,'L',3,2),(3,2,'O',1,1),(1,1,'!',2,2)\}$$

The output may not be sorted as shown above. The list can be obtained by walking the linked list starting from the sentinel root element with id $(0,0)$ and collecting the values along the way which yields the text "HELLO!".

To demonstrate the nature of incremental computations and the deletion of elements, I continue the example with two successive deletions of elements and show the output *changes* after each deletion. Output changes are encoded as tagged tuples as described in Section 2.3. The two deletions are applied on top of the state of the list from Figure 4.4. In the first step, the last element "!" is deleted, which adds the fact "remove(2,2)" to the "remove" EDBP. This yields the output change:

$$\{(-1,(1,1,'!',2,2))\}$$

The first entry of the tagged tuple is the $\mathbb{Z}$-weight of -1 and indicates a deletion of the tuple $(1,1,'!',2,2)$ (second entry of the pair) from the previous list state. This yields the text "HELLO".

The second step deletes the first element "H" and causes the fact "remove(2,1)" to be inserted into the "remove" EDBP. A reevaluation then emits the following output changes:

$$\{(-1,(0,0,'H',2,1)),(-1,(2,1,'E',2,3)),(1,(0,0,'E',2,3))\}$$

The first two changes retract two tuples from the previous output state and the third change adds a new tuple because of its $\mathbb{Z}$-weight of $+1$. Due to the nature of linked lists, any deletion of an element, which is *not* the last element, triggers three output changes: One for the deletion of the element itself and two for replacing the previous "pointer" of the deleted element's successor in the list. The final text is now "ELLO".

4.2 Performance Evaluation

Section 4.2.1 and Section 4.2.2 provide benchmarks of the three CRDTs from the previous section. Benchmarking happens in two different settings which are distinguished by the operators of the query plan being either "cold" or "warm":

- **Hydration setting** (Section 4.2.1). As outlined in Section 2.3, all bilinear operators are stateful. Hence, all stateful operators of the query plan have to restore their state before processing any new updates, e.g., in case of a "cold" application restart. This benchmark measures the time it takes until the application can show its last state and process *new* updates again on top of an existing operation history.
- **Near-real-time setting** (Section 4.2.2). In this setting, the operators are assumed to be "warm" and only a small number of updates are new to them. This benchmark measures the common operation mode of CRDTs after the application has started up. It measures how much time it takes to process a small number of updates on top of an already "hydrated" query plan.

All benchmarks are executed on a 2024 MacBook Pro with an Apple M4 Pro CPU. DBSP's runtime is configured to run in a single thread due to correctness issues with its multi-threaded execution for queries that use recursion. Although I am not sure

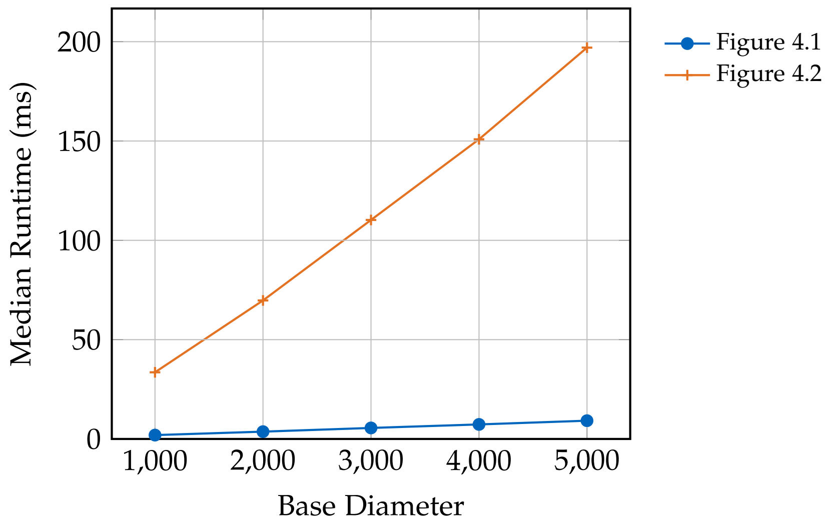


Figure 4.5: Hydration setting benchmark for the key-value stores.

about the exact reason, I suspect that this is because some of DBSP's operators are not thread-safe, as mentioned in their documentation. Unfortunately, it currently lacks a list of non-thread-safe operators and does not explain the issue any further[1].

4.2.1 Hydration Setting

As the purpose of the hydration setting is to model application startups, it includes the time to parse the CRDT Datalog query and set up the query plan, i.e., the time spent during query build-time, in addition to processing an existing causal history.

The benchmark for the key-value stores in the hydration setting is set up as follows. The *base diameter* $\in \{1000, 2000, 3000, 4000, 5000\}$ is the diameter of the causal history if viewed as a directed graph, i.e., it is the longest shortest path from a root to a leaf. As the causal history is a chain of operations with no concurrency, the base diameter is equal to the number of operations (plus one). All operations are generated at one replica and write to the same key. The recorded operations during that process are fed into the DBSP circuit after it has been constructed.

Figure 4.5 shows the results of both the key-value store without causal broadcast from Figure 4.1 and the key-value store including causal broadcast from Figure 4.2. The causal broadcast mechanism has a significant impact on the performance: While an increase of the base diameter by 1000 causes the performance of the key-value store without causal broadcast to increase by roughly two milliseconds, the key-value

[1]`https://docs.rs/dbsp/0.64.0/dbsp/circuit/struct.Runtime.html#method.init_circuit`

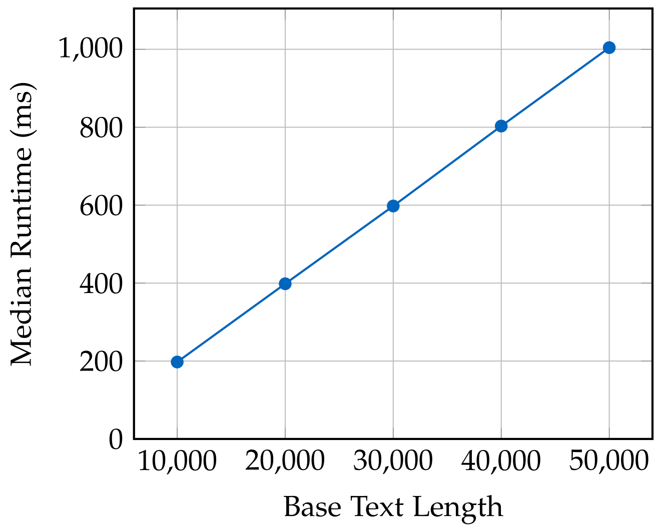


Figure 4.6: Hydration setting benchmark for the list CRDT from Figure 4.3.

store's performance including causal broadcast increases by about 40 milliseconds. The reason for this behavior is the graph traversal of the causal history from its roots to its leaves to find the causally ready operations. The fixed point iteration responsible for this graph traversal requires evaluating a sequence of *dependent* joins whose length is relative to the diameter of the causal history.

Figure 4.6 shows the benchmark results of the list CRDT. The benchmark generates a causal history of successive insert operations, i.e., there are no deletions and cursor jumps but only consecutive insertions that gradually build up the text by inserting new characters at the respective ends of the list. Hence, the *base text length* $\in \{10000, 20000, 30000, 40000, 50000\}$ is the number of insert operations. The benchmark shows that an increase in the base text length by 10000 insertions causes an increase in the runtime of about 200 milliseconds. Loading a document with 50000 operations in this scenario takes about a second.

4.2.2 Near-Real-Time Setting

The near-real-time setting extends the hydration setting insofar that it builds upon the states of the hydration setting. After the operators have been hydrated according to the parameters of the hydration setting, this benchmark feeds in only a small number of additional updates. The batch of new updates is inserted at once and then processed by the DBSP circuit to obtain the CRDT state changes. The benchmark does not account for the time it takes to parse the Datalog query nor to set up the DBSP circuit, as the near-real-time setting simulates the situation in which the application has already started up and is running.

Figure 4.7 shows benchmark results of the key-value stores. A *delta diameter* of size $d \in \{20, 40, 60, 80, 100\}$ extends the chain of operations from the hydration setting by $d + 1$ additional operations. The benchmark reveals some interesting patterns. For the key-value store without causal broadcast from Figure 4.1, the base diameter has no real impact on the performance. Varying the delta diameter has only a small impact on the performance. In any case, the processing of the updates takes less than a quarter of a millisecond (Figure 4.7a). The key-value store including causal broadcast from Figure 4.2 also shows only a small performance impact from varying the delta diameter. The base diameter, on the other hand, has a significant impact on the performance (Figure 4.7b): An increase in the base diameter by 1000 causes an increase of the runtime of about 20 milliseconds for all delta diameters, rendering this CRDT *dependent* on the age of the causal history. This is again due to the dependent joins of the causal broadcast's fixed point iteration. I suspect that even a multi-threaded execution would not improve the performance, as the joins are dependent, rendering the computation inherently sequential. However, a clever query optimizer could possibly turn the causal broadcast into a near constant time operation by, e.g., testing causal readiness of a new operation from the current leaves of the causal history and thus avoiding the costly search from the roots, as most new operations are likely a causal successor of the current leaves or nodes close to them.

Figure 4.8 shows the benchmark results of the list CRDT. A *delta text length* of size $d \in \{20, 40, 60, 80, 100\}$ adds a burst of d successive insertions at the respective ends of the list. Applying the burst and evaluating the list CRDT takes between 1.5 and 5.25 milliseconds depending on the base text length and the delta text length. Similar to the key-value store including causal broadcast, the base text length has an impact on the performance, albeit not as significant. A base text length increase of 10000 causes an increase in the runtime by roughly half a millisecond for any delta text length. Increasing the delta text length by 20 causes an increase in the runtime by about 0.4 milliseconds for all base text lengths.

4.3 Related Work

The idea of using restricted languages to express CRDTs to guarantee their convergence has been explored in the past. VeriFx [6], Propel [47], and LoRe [3] introduce custom DSLs to define CRDTs. From a CRDT definition in the DSL, they derive both its implementation and its proof of convergence. The verification is handed off to a SAT solver, which is used to prove that the CRDT converges under concurrent updates.

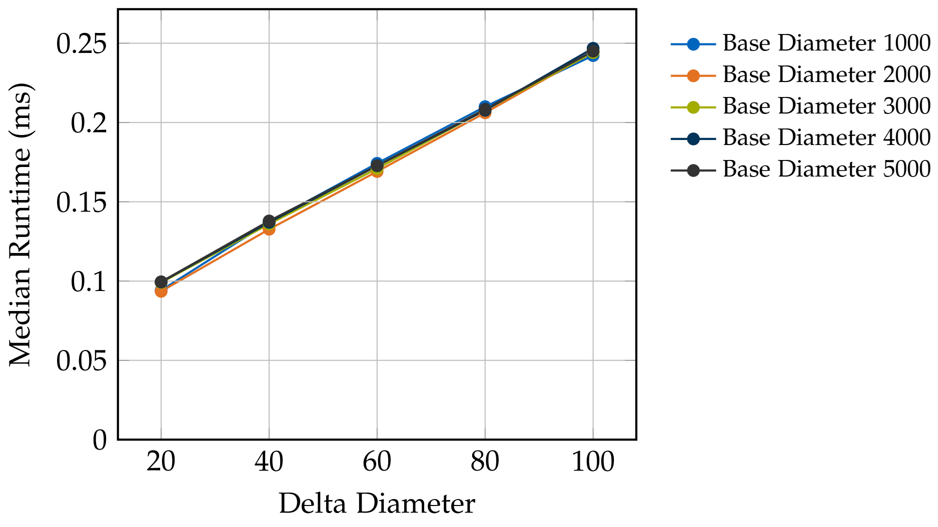
LoRe differs from VeriFx insofar that it can also express invariants that require coordination between replicas and inject such coordination logic automatically, freeing the application developer from writing synchronization code. In contrast to DSLs, Datalog has the advantage that it is a more widely used language and does not require verification time because convergence is guaranteed by construction if used on top of monotonically growing input sets.

There exist several libraries that implement CRDTs in general-purpose programming languages, for instance, Automerge [1] (Rust), Yjs [2] (JavaScript), Collabs [48] (TypeScript), and Loro [49] (Rust). They come with no formal proof of convergence but are crafted by experienced developers familiar with eventual consistency and are widely used in practice. Yet, changing them to support custom data types is a challenging task and requires a deep understanding of the library's internals as well as theoretical foundations of CRDTs, rendering them less flexible than language based approaches.

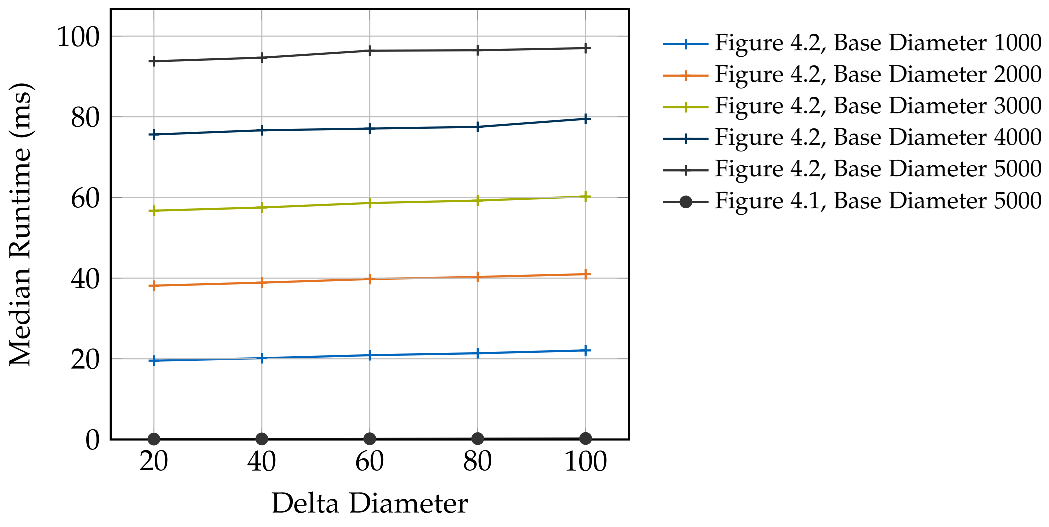
Other CRDT frameworks are based on (immutable) event logs, deriving state from the log, and replaying events in a consistent order across replicas to handle concurrent updates and/or updates which are delivered out-of-order. The idea has been formalized in [24] and is applied in the Eg-walker [50] algorithm in the context of collaborative text editing. LiveStore [51] is a recent framework, which allows developers to specify events and how to derive state from them. If the derivation happens deterministically, convergence is guaranteed, too, due to the consistent event order across replicas. Yet, the event order is unstable in case of concurrent or out-of-order events, as they require undoing and (re)applying events. This renders supporting list operations challenging because the list operations' indexes may need to be transformed to match the different list order based on the new event order.

In general, there are two predominant approaches to query execution, *interpreted* and *compiled* query execution. The former executes a query by interpreting its query plan at run-time, usually on the granularity level of an operator of the query plan. Compiled query execution has been pioneered by the HyPer main memory database system [52]. It compiles a query plan into an executable tailored to the specific query, can therefore take advantage of, e.g., combining multiple non-blocking operators into a single loop, and avoids interpretation overhead. While compiled query execution sounds more promising in terms of performance, it is more complex to implement, debug, and, surprisingly, its performance is not necessarily better [53]. Interestingly, DBSP's repository includes a SQL-to-DBSP compiler which emits a Rust executable to execute a specific query [36]. The now-unmaintained Differential Datalog project [54], [55], which relies on differential dataflow [34], also compiles a query plan into a Rust executable. The strong preference for compiled query execution is unclear to me and, as stated in Section 5.1, I am interested in seeing their differences being analyzed more closely, especially in the context of *incremental* query engines. Only [56] explores executing Datalog queries incrementally on top of DBSP through an interpreter but it only supports positive Datalog. In comparison to my approach, it allows adding and removing rules dynamically at run-time but it is more tied to DBSP because the Datalog evaluation happens directly on a DBSP circuit without an IR as an abstraction.

There are two predominant approaches for incremental computations beyond the bag algebra approach: Either they are based on differential dataflow [34] and timely dataflow [57], or they are based on the DBSP [11], [35]. Both approaches are backed by a commercial offering, Feldera [58] and Materialize [59], respectively. A prominent representative of non-incremental Datalog engines is Soufflé [60] which compiles Datalog queries into C++ code to execute them efficiently. Ascent [61] allows seamlessly embedding Datalog queries into Rust programs by defining them through Rust's macros. Flix [62] extends Datalog's semantics with support for lattices to facilitate expressing programs to solve problems in static program analysis. Datafrog [63] is a Rust library that does not implement a Datalog dialect but provides library functions to express Datalog-style rules and execute them in the context of the calling thread.



(a) Key-value store without causal broadcast from Figure 4.1.



(b) Key-value store including causal broadcast from Figure 4.2. The measurement of the key-value store *without* causal broadcast for the worst case is shown for reference.

Figure 4.7: Near-real-time setting benchmark for the key-value stores.

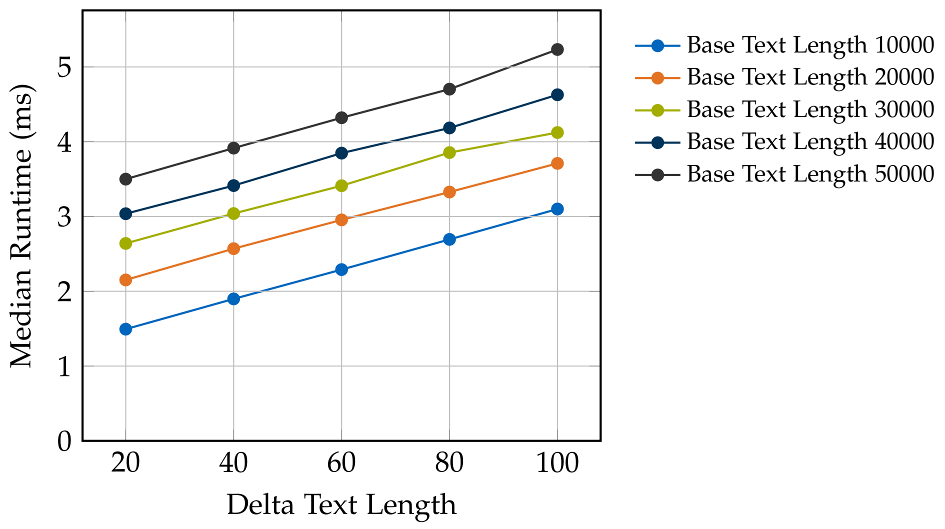


Figure 4.8: Near-real-time setting benchmark for the list CRDT from Figure 4.3.

5 Conclusion

5.1 Future Work

In this work, I took several technical decisions to have a working prototype available but many of them leave open questions for future work. First, this work chooses DBSP as the IVM framework for the query engine. This leaves the question how other IVM frameworks, such as differential dataflow [34], compare to DBSP in their performance and expressiveness for this use case. Second, it is an open question if (and by what margin) compiling query plans outperforms interpreting them in the context of IVM. Third, what benefit could a mixed model of incremental and non-incremental computations provide: Possibly, it may be faster to prefer non-incremental execution during application startup (the hydration setting; Section 4.2.1) and then switch to incremental execution for processing updates (the near-real-time setting; Section 4.2.2). The required hydration of the operators for the incremental execution could be moved to a background thread. Alternatively, the operators' state could somehow be persisted to disk such that the query engine can resume from a previous state and does not have to feed in all updates again, in which case the query engine may not need to support a non-incremental mode at all.

To minimize the interpretation overhead of the query engine, it could benefit from some performance engineering. The physical representation of a tuple is currently a vector of scalar values. Hence, accessing a tuple's fields requires dereferencing a pointer and thereby potentially causing a cache miss with each (first) field access. By inlining so many fields into the tuple representation that a cache line is filled, the number of cache misses can be reduced. If necessary, further fields could overflow to the heap[1]. To further reduce the number of cache misses, the physical representation of an IR program could be adapted to use a more cache-friendly data layout, too: Instead of expressions storing references to other expressions in their input fields, they could be flattened by storing an index into a vector of expressions, as laid out in [64]. This may improve spatial locality, thereby reducing cache misses, and is particularly relevant for code that is run at query execution-time because it is executed frequently (per tuple) instead of just once at query build-time for initializing the DBSP circuit.

[1]Rust's `smallvec` library implements this concept.

Applying serious query optimization has the potential to improve the performance of the query engine but this is a challenge in its own right. Particularly, in the context of IVM and translation from Datalog to relational algebra, I want to emphasize three aspects:

- **Join ordering**. As discussed in Section 2.3, changing a query plan for a continuously evaluated query can be costly. Also, the query plan has to be chosen upfront and with the absence of statistics about the data. Hence, to what extent is an automated join ordering algorithm useful in this context? Possibly, it may be better to let the query author choose a definitive join order or allow them to provide information about data distributions in case of automated join ordering. Alternatively, the query engine could remember statistics from one application run and use them to optimize the query plan for its next run when the query plan needs to be reconstructed anyways.
- **Scheduling of antijoins**. Negative atoms are handled with antijoins in the query engine. So far, they are scheduled after the query plan covering the positive atoms. However, antijoins may actually be scheduled better by placing them as early as possible, i.e., if all variables referenced by a negative atom are in scope, to keep the intermediate result set small. This is because in the worst case, antijoins leave the intermediate result set unchanged but in the average case they filter tuples out. This is similar to predicate pushdown and, therefore, I call this problem *antijoin pushdown*.
- **Query optimization with Datalog**. As this approach uses both Datalog and relational algebra, there are two abstraction levels upon which query optimization can be applied. Although query optimization on Datalog has been studied to a lesser extent, it may offer other, unexplored possibilities which can only (or better) be applied on the Datalog level. Also, this requires researching the interaction between Datalog and relational algebra optimizations.

These aspects represent open ends to the bigger question if a query based approach to CRDTs can be made competitive in performance with hand-written CRDTs defined in general-purpose programming languages. If they lack in terms of performance, by what margin do they fall short? In other words, what is the price to pay for guaranteed convergence?

Feature-wise, the query engine could be extended to support mutual recursion and aggregation but I do not consider these features essential for the CRDT use case. Native JSON support may be useful for defining a JSON CRDT like Automerge does. As motivated in Section 3.3, supporting multiple main predicates of interest in a single query is not yet implemented but useful in practice. To improve practical usability of the Datalog frontend, a type checking pass, better error reporting, and supporting tuples as a scalar type could be added to the query engine. The latter allows collapsing operations' replica id and counter fields into a single field, to make Datalog CRDT queries more concise.

To use this approach in practice, some more engineering effort is required. The query engine is currently just a computation engine and does not provide a durable storage layer for the underlying data. Additionally, it remains unclear how compatible this approach is with partial synchronization and update compaction, to keep storage sizes manageable on smaller edge devices which cannot afford to store a monotonically growing set of updates.

Section 4.1 defines map and list CRDTs as queries. More research is needed to define and benchmark further CRDTs as queries before putting this idea into practice. It is especially important to better understand the performance under various workloads, e.g., different concurrency and usage patterns, to guide future directions for optimizing the query engine. A sophisticated simulation framework which can not only model concurrency among replicas but also network delays, and allows writing adapters to different CRDT implementations would be useful for testing and benchmarking different CRDTs. Finally, the question remains if there are CRDTs that cannot be expressed with Datalog's restricted expressiveness with stratified negation. A move operation in replicated trees [65], [66] may be a challenging candidate, as it requires rolling back updates to a previous state, applying the move operation, then reapplying the previously undone updates, and potentially skipping operations if executing them would introduce a cycle.

5.2 Outlook

Similar to how queries made data retrieval and storage more accessible to application developers, I hope that expressing CRDTs as queries makes CRDTs more accessible to a wider audience, too. This approach has the potential to simplify the development of collaborative, local-first applications [67], by allowing developers to focus on the application logic instead of the underlying problems in eventually consistent, distributed systems, while still providing the possibility to fine-tune a CRDT's behavior.

Furthermore, I hope that this work inspires others to seriously consider Datalog as an alternative to SQL for a query language due to its succinct syntax and solid foundation in logic programming.

Abbreviations

CRDT convergent replicated data type

IR intermediate representation

AST abstract syntax tree

EDBP extensional database predicate

IDBP intensional database predicate

MVR multi-valued register

LWWR last-writer-wins register

IVM incremental view maintenance

GUI graphical user interface

API application programming interface

SAT satisfiability

DSL domain-specific language

RGA replicated growable arrays

List of Figures

Bibliography

[1] The Automerge Contributors, *Automerge CRDT Library*, `https://github.com/automerge`, 2024.

[2] The Yjs Contributors, *Yjs CRDT Library*, `https://github.com/yjs/yjs`, 2025.

[3] J. Haas, R. Mogk, E. Yanakieva, A. Bieniusa, and M. Mezini, "LoRe: A programming model for verifiably safe local-first software," *ACM Trans. Program. Lang. Syst.*, vol. 46, no. 1, Jan. 2024, ISSN: 0164-0925. DOI: `10.1145/3633769`.

[4] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford, "Verifying strong eventual consistency in distributed systems," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017. DOI: `10.1145/3133933`.

[5] M. Kleppmann, "Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-969, May 2022. DOI: `10.48456/tr-969`.

[6] K. De Porre, C. Ferreira, and E. Gonzalez Boix, "VeriFx: Correct Replicated Data Types for the Masses," in *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, K. Ali and G. Salvaneschi, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 263, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 9:1–9:45, ISBN: 978-3-95977-281-5. DOI: `10.4230/LIPIcs.ECOOP.2023.9`.

[7] M. Weidner and M. Kleppmann, *The art of the fugue: Minimizing interleaving in collaborative text editing*, 2023. arXiv: `2305.00583 [cs.DC]`.

[8] M. Kleppmann. "Data structures as queries: Expressing CRDTs using Datalog." (2018), [Online]. Available: `https://martin.kleppmann.com/2018/02/26/dagstuhl-data-consistency.html`.

[9] L. Yang, Y. Gilad, and M. Alizadeh, "Practical rateless set reconciliation," in *Proceedings of the ACM SIGCOMM 2024 Conference*, ser. ACM SIGCOMM '24, Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 595–612, ISBN: 9798400706141. DOI: `10.1145/3651890.3672219`.

[10] A. Meyer, *Range-based set reconciliation*, 2023. arXiv: `2212.13567 [cs.CR]`.

[11] M. Budiu, L. Ryzhyk, G. Zellweger, B. Pfaff, L. Suresh, S. Kassing, A. Gyawali, M. Budiu, T. Chajed, F. McSherry, and V. Tannen, "DBSP: Automatic incremental view maintenance for rich query languages," *The VLDB Journal*, vol. 34, no. 4, May 2025, ISSN: 1066-8888. DOI: `10.1007/s00778-025-00922-y`.

[12] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 185–196, Nov. 2014, ISSN: 2150-8097. DOI: `10.14778/2735508.2735509`.

[13] DittoLive Inc. "DittoLive's homepage." (2025), [Online]. Available: `https://web.archive.org/web/20250613135524/https://www.ditto.com/` (visited on 06/13/2025).

[14] G. Litt, N. Schiefer, J. Schickling, and D. Jackson, "Riffle: Reactive relational state for local-first applications," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23, San Francisco, CA, USA: Association for Computing Machinery, 2023, ISBN: 9798400701320. DOI: `10.1145/3586183.3606801`.

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, ISSN: 0001-0782. DOI: `10.1145/359545.359563`.

[16] M. Abo Khamis, H. Q. Ngo, R. Pichler, D. Suciu, and Y. R. Wang, "Convergence of datalog over (pre-) semirings," *J. ACM*, vol. 71, no. 2, Apr. 2024, ISSN: 0004-5411. DOI: `10.1145/3643027`.

[17] T. Neumann and V. Leis, "A critique of modern SQL and a proposal towards a simple and expressive query language," in *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*, www.cidrdb.org, 2024.

[18] D. Hirn and T. Grust, "A fix for the fixation on fixpoints," in *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*, www.cidrdb.org, 2023.

[19] F. McSherry. "Recursion in Materialize." (2022), [Online]. Available: `https://web.archive.org/web/20241126143413/https://github.com/frankmcsherry/blog/blob/master/posts/2022-12-25.md` (visited on 11/26/2024).

[20] T. J. Green, S. S. Huang, B. T. Loo, and W. Zhou, "Datalog and recursive query processing," *Foundations and Trends Databases*, vol. 5, no. 2, pp. 105–195, Nov. 2013, ISSN: 1931-7883. DOI: `10.1561/1900000017`.

[21] M. A. Khamis, H. Q. Ngo, R. Pichler, D. Suciu, and Y. Remy Wang, "Datalog in wonderland," *SIGMOD Rec.*, vol. 51, no. 2, pp. 6–17, Jul. 2022, ISSN: 0163-5808. DOI: `10.1145/3552490.3552492`.

[22] K. R. Apt, H. A. Blair, and A. Walker, "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 89–148, ISBN: 0934613400.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011, p. 50.

[24] C. Baquero, P. S. Almeida, and A. Shoker, "Pure operation-based replicated data types," *CoRR*, vol. abs/1710.04469, 2017. arXiv: `1710.04469`.

[25] L. Stewen and M. Kleppmann, "Undo and redo support for replicated registers," in *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '24, Athens, Greece: Association for Computing Machinery, 2024, pp. 1–7, ISBN: 9798400705441. DOI: `10.1145/3642976.3653029`.

[26] A. Auvolat and F. Taïani, "Merkle search trees: Efficient state-based CRDTs in open networks," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 221–22 109. DOI: `10.1109/SRDS47363.2019.00032`.

[27] H. Sanjuan, S. Poyhtari, P. Teixeira, and I. Psaras, "Merkle-CRDTs: Merkle-DAGs meet CRDTs," *CoRR*, vol. abs/2004.00107, 2020. arXiv: `2004.00107`.

[28] M. Kleppmann, P. Frazee, J. Gold, J. Graber, D. Holmgren, D. Ivy, J. Johnson, B. Newbold, and J. Volpert, "Bluesky and the at protocol: Usable decentralized social media," in *Proceedings of the ACM Conext-2024 Workshop on the Decentralization of the Internet*, ser. DIN '24, Los Angeles, CA, USA: Association for Computing Machinery, 2024, pp. 1–7, ISBN: 9798400712524. DOI: `10.1145/3694809.3700740`.

[29] M. Kleppmann, "Making CRDTs Byzantine fault tolerant," in *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '22, Rennes, France: Association for Computing Machinery, 2022, pp. 8–15, ISBN: 9781450392563. DOI: `10.1145/3517209.3524042`.

[30] A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," in *Materialized Views: Techniques, Implementations, and Applications*. Cambridge, MA, USA: MIT Press, 1999, pp. 145–157, ISBN: 0262571226.

[31] A. Gupta, I. S. Mumick, and V. S. Subrahmanian, "Maintaining views incrementally," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93, Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 157–166, ISBN: 0897915925. DOI: `10.1145/170035.170066`.

[32] The pg_ivm Contributors, *pg_ivm PostgreSQL Extension*, `https://github.com/sraoss/pg_ivm`, 2025.

[33] M. Idris, M. Ugarte, and S. Vansummeren, "The Dynamic Yannakakis algorithm: Compact and efficient query processing under updates," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1259–1274, ISBN: 9781450341974. DOI: `10.1145/3035918.3064027`.

[34] F. McSherry, D. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *Proceedings of CIDR 2013*, Jan. 2013.

[35] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen, "DBSP: Incremental computation on streams and its applications to databases," *SIGMOD Rec.*, vol. 53, no. 1, pp. 87–95, May 2024, ISSN: 0163-5808. DOI: `10.1145/3665252.3665271`.

[36] The Feldera Contributors, *Feldera and DBSP Library*, `https://github.com/feldera/feldera`, 2025.

[37] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '07, Beijing, China: Association for Computing Machinery, 2007, pp. 31–40, ISBN: 9781595936851. DOI: `10.1145/1265530.1265535`.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34, ISBN: 089791001X. DOI: `10.1145/582095.582099`.

[39] R. Nystrom, *Crafting interpreters*. Genever Benning, 2021.

[40] The Nom Contributors, *Nom Parser Combinator Framework*, `https://github.com/rust-bakery/nom`, 2025.

[41] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962, ISSN: 0001-0782. DOI: `10.1145/368996.369025`.

[42] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 395–403. DOI: `10.1109/ICDCS.2009.20`.

[43] S. Weiss, P. Urso, and P. Molli, "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks," in *2009 29th IEEE International Conference on Distributed Computing Systems*, 2009, pp. 404–412. DOI: `10.1109/ICDCS.2009.75`.

[44] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *J. Parallel Distrib. Comput.*, vol. 71, no. 3, pp. 354–368, Mar. 2011, ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2010.12.006`.

[45] V. Grishchenko, "Deep hypertext with embedded revision control implemented in regular expressions," in *Proceedings of the 6th International Symposium on Wikis and Open Collaboration*, ser. WikiSym '10, Gdansk, Poland: Association for Computing Machinery, 2010, ISBN: 9781450300568. DOI: `10.1145/1832772.1832777`.

[46] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, "Specification and complexity of collaborative text editing," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, ser. PODC '16, Chicago, Illinois, USA: Association for Computing Machinery, 2016, pp. 259–268, ISBN: 9781450339643. DOI: `10.1145/2933057.2933090`.

[47] G. Zakhour, P. Weisenburger, and G. Salvaneschi, "Type-checking CRDT convergence," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. DOI: `10.1145/3591276`.

[48] The Collabs Contributors, *Collabs CRDT Library*, `https://github.com/composablesys/collabs`, 2025.

[49] The Loro Contributors, *Loro CRDT Library*, `https://github.com/loro-dev/loro`, 2025.

[50] J. Gentle and M. Kleppmann, "Collaborative text editing with eg-walker: Better, faster, smaller," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25, Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 311–328, ISBN: 9798400711961. DOI: `10.1145/3689031.3696076`.

[51] The LiveStore Contributors, *LiveStore Library*, `https://github.com/livestorejs/livestore`, 2025.

[52] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, Jun. 2011, ISSN: 2150-8097. DOI: `10.14778/2002938.2002940`.

[53] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proc. VLDB Endow.*, vol. 11, no. 13, pp. 2209–2222, Sep. 2018, ISSN: 2150-8097. DOI: `10.14778/3275366.3284966`.

[54] The Differential Datalog Contributors, *Differential datalog*, `https://github.com/vmware-archive/differential-datalog`, 2025.

[55] L. Ryzhyk and M. Budiu, "Differential datalog," in *Datalog 2.0*, Philadelphia, PA, Jun. 2019.

[56] B. R. C. A. de Lima, K. Apinis, M. Kramer, and K. K. Micinski, "Incremental evaluation of dynamic datalog programs as a higher-order DBSP program," in *Proceedings 5th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2024) co-located with the 17th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2024), Dallas, Texas, USA, October 11, 2024*, M. Alviano and M. Lanzinger, Eds., ser. CEUR Workshop Proceedings, vol. 3801, CEUR-WS.org, 2024, pp. 2–16.

[57] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 439–455, ISBN: 9781450323888. DOI: `10.1145/2517349.2522738`.

[58] Feldera, Inc. "Feldera's homepage." (2025), [Online]. Available: `https://web.archive.org/web/20250714140906/https://www.feldera.com/` (visited on 07/14/2025).

[59] Materialize, Inc. "Materialize's homepage." (2025), [Online]. Available: `https://web.archive.org/web/20250714140957/https://materialize.com/` (visited on 07/14/2025).

[60] B. Scholz, H. Jordan, P. Suboti, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16, Barcelona, Spain: Association for Computing Machinery, 2016, pp. 196–206, ISBN: 9781450342414. DOI: `10.1145/2892208.2892226`.

[61] A. Sahebolamri, T. Gilray, and K. Micinski, "Seamless deductive inference via macros," in *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2022, Seoul, South Korea: Association for Computing Machinery, 2022, pp. 77–88, ISBN: 9781450391832. DOI: `10.1145/3497776.3517779`.

[62] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," *SIGPLAN Not.*, vol. 51, no. 6, pp. 194–208, Jun. 2016, ISSN: 0362-1340. DOI: 10.1145/2980983.2908096.

[63] The Datafrog Contributors, *Datafrog*, https://github.com/rust-lang/datafrog, 2025.

[64] A. Sampson. "Flattening ASTs (and Other Compiler Data Structures)." (2023), [Online]. Available: https://web.archive.org/web/20250613165634/https://www.cs.cornell.edu/~asampson/blog/flattening.html (visited on 07/03/2025).

[65] M. Kleppmann, D. P. Mulligan, V. B. F. Gomes, and A. R. Beresford, "A highly-available move operation for replicated trees," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 7, pp. 1711–1724, 2022. DOI: 10.1109/TPDS.2021.3118603.

[66] L. Da and M. Kleppmann, "Extending JSON CRDTs with move operations," in *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '24, Athens, Greece: Association for Computing Machinery, 2024, pp. 8–14, ISBN: 9798400705441. DOI: 10.1145/3642976.3653030.

[67] M. Kleppmann, A. Wiggins, P. van Hardenberg, and M. McGranaghan, "Local-first software: You own your data, in spite of the cloud," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019, Athens, Greece: Association for Computing Machinery, 2019, pp. 154–178, ISBN: 9781450369954. DOI: 10.1145/3359591.3359737.