Invited Paper: Initial Steps Toward a Compiler for Distributed Programs

Joseph M. Hellerstein
hellerstein@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Shadaj Laddad
shadaj@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Mae Milano
mpmilano@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Conor Power
conorpower@berkeley.edu
UC Berkeley
Berkeley, CA, USA

Mingwei Samuel
mingwei@shv.com
Sutter Hill Ventures
Palo Alto, CA, USA

ABSTRACT

In the Hydro project we are designing a compiler toolkit that can optimize for the concerns of distributed systems, including scale-up and scale-down, availability, and consistency of outcomes across replicas. This invited paper overviews the project, and provides an early walk-through of the kind of optimization that is possible. We illustrate how type transformations as well as local program transformations can combine, step by step, to convert a single-node program into a variety of distributed design points that offer the same semantics with different performance and deployment characteristics.

KEYWORDS

distributed computing, programming languages, compiler, query optimization, dataflow

ACM Reference Format:
Joseph M. Hellerstein, Shadaj Laddad, Mae Milano, Conor Power, and Mingwei Samuel. 2023. Invited Paper: Initial Steps Toward a Compiler for Distributed Programs. In *The 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED 2023), June 19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3584684.3597272

1 INTRODUCTION

An ongoing thread in distributed computing is the development of new programming models and languages built on formal models that can simplify the challenges developers face in building distributed software. This thread includes programming languages like Dedalus [3], Bloom [2, 10], LVars [27], Lasp [32], Datafun [4] and Gallifrey [33], as well as data structures like CRDTs [38] and their realization in libraries like Automerge [24].

This short paper is part of an evolution from language design to a full stack for distributed programming. Is it possible to build a language stack (multiple surface languages with shared compilation, debugging and deployment) that addresses the concerns of developers writing distributed programs? How much work can be done automatically? Of what remains, what is amenable to compiler assistance and human review? Can compiled code compete with hand-written code? Can a compiler discover optimizations that humans do not? This paper is an early snapshot of our work in this domain. It does not claim to answer all of these questions, nor even to answer any one of them definitively. We narrow our focus here to automatic compiler transformations, with much of the discussion driven from simple examples. In short, this paper is intended as a progress report and an opening for community engagement.

Traditional optimizing compilers concern themselves with efficient use of computing resources including the various aspects of CPUs, GPUs, memory and interconnects. All of these concerns exist in distributed systems of course, but are augmented by concerns that are endemic to distributed systems: notably communication, partitioning of work across machines, fault tolerance, concurrency and consistency of data and outcomes, and respect for invariants related to security, privacy and governance. We assume that traditional optimizer toolkits like LLVM [30] will continue to serve the purposes of optimizing the code that runs on each machine; we focus on the challenge of providing abstractions and compilers for the unique distributed aspects of modern programs.

We ground our discussion in the context of the Hydro project, a multi-year effort at UC Berkeley and Sutter Hill Ventures. We laid out our vision for Hydro in an earlier paper [8], proposing a compiler stack (Figure 1) with multiple components. As the highest level of input to Hydro, we hope to support a multitude of distributed programming styles, much like LLVM provides a compiler stack for a multitude of sequential programming languages. Also like LLVM, Hydro envisions multiple layers of intermediate representation languages (IRs) that can serve as common ground for program checks and transformations, providing developers with various entry points to work deeper in the stack as they see fit. The top layer of Hydro we call *Hydraulic*: a system to lift low-level code from legacy interfaces into a higher-level declarative distributed IR we dub *Hydrologic*. The next layer down is an optimizing compiler we call *Hydrolysis*, which takes Hydrologic and compiles it to run on multiple instances of a single-threaded, asynchronous dataflow

This work is licensed under a Creative Commons Attribution International 4.0 License.

ApPLIED 2023, June 19, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0128-3/23/06.
https://doi.org/10.1145/3584684.3597272

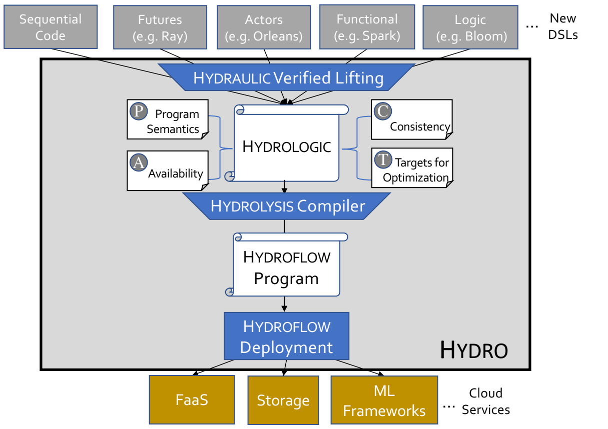

Figure 1: The Hydro compiler stack [8].

IR called *Hydroflow*. Work toward Hydraulic and Hydrologic is ongoing, with initial results beginning to emerge [28].

The focus for this paper is on Hydrolysis—specifically we want to explore the feasibility of a compiler that can produce Hydroflow programs optimized for different deployment objectives. We consider the potential for a *transformation-based* compiler that can correctly modify a Hydroflow program, step-by-step, into one or more alternative Hydroflow programs that offer the same results with different performance or deployment characteristics. Like many dataflow systems, Hydroflow is an extension of the relational algebra. As such it is amenable to the kinds of query optimizations pioneered for database systems, as well as a host of optimizations that are apropos for general-purpose distributed programs. Following the model of the widely-used Cascades query optimizer [15] and the renewed enthusiasm in the programming languages community for e-graphs [35, 41], we view query optimization as a problem of *program transformation*. Basically, we assume we can translate from Hydrologic to *some* semantically-equivalent *single-node* Hydroflow program (“execution plan”) in a naive fashion—essentially via parsing Hydrologic without concern for distributed deployment or performance. The compiler’s job then is to search the space of semantically equivalent dataflow programs running on one or more machines, and choose the configuration that is most desirable according to an objective function on various metrics (e.g. cloud costs, runtime performance, resilience to failures, etc.) with various constraints or invariants on how data and communication are to be managed.

In the style of Cascades and egraphs, we envision optimization via repeated application of simple local program transformations, using compact memoization to keep track of prior states and avoid repeated work. The basic loop is this: the current Hydroflow program is stored in a memoization structure, and a local *transformation rule* (a kind of peephole optimization) is applied to a small segment of the Hydroflow program to get another semantically equivalent Hydroflow program that we have not seen before. In the presence of a cost model and objective function, pruning is applied to the memo structure and search space to keep only those candidates that may participate in an optimal outcome. This process repeats until all possible semantically equivalent dataflows have been generated or ruled out as suboptimal via pruning.

This paper does not deliver on the full vision of the Hydrolysis optimizer, though we sketch next steps in Section 6. Our discussion here is an early demonstration of manually applied transformation rules that achieve useful optimizations for correctly distributing programs across machines. We do not pretend to offer a comprehensive set of such transformations as of yet. Our goal is to document our growing confidence in the potential for an optimizing compiler—specifically one based in a dataflow model—to meaningfully assist in the development of efficient, correct distributed programs.

1.1 Why Dataflow?

One of the signatures of the Hydro project is the opinionated decision to use a high-performance dataflow kernel as its lowest-level language. This may not be an obvious choice to researchers in classical distributed systems.

At the outset, we were confident about the scalability of a dataflow IR because of the success of prior dataflow engines at auto-parallelization. Languages like SQL and Spark have put effortless scaling into the hands of programmers for decades, even as other broad efforts at parallel and distributed programming languages failed [19]. The runtimes for those data-centric languages are parallelized dataflow engines. Unlike Spark and Hadoop, dataflow runtimes for SQL have targeted heterogeneous workloads and performance goals, including low-latency infrastructure. There was reason to be confident that dataflow can meet our low-level performance goals *and* scale with ease, as we discuss below.

More generally, by using dataflow we gained access to a long tradition of database and compiler literature on optimization. The database literature is founded on the duality between dataflow algebra and high-level query languages like SQL—i.e., Codd’s Theorem [9], the basis of his Turing Award. Because Hydroflow is so close to query languages like SQL or Datalog, we can apply the full body of database theory and practice to our compiler runtime.

One of the benefits of dataflow and query languages that we exploit is the ease of refactoring code. Auto-parallelization of sequential code involves teasing apart a monolithic program into separable components, and dataflow makes this almost trivial. Every dataflow program is a graph of producers and consumers, so refactoring a program into separate software components is almost as simple as changing local dataflow pipes into network channels. Of course this requires care to maintain program semantics, as we’ll discuss below. By contrast, as any software engineer knows, it can be very hard to refactor a sequential program without breaking it—this is especially true of complex programs like the Paxos variants we have been building.

Second, the primary syntactic feature of a dataflow language is the explicit *specification of data dependencies*. Our transformations use data dependencies to analyze the interplay between components, and reason about the implications of placing components on separate nodes across networks. By contrast, data dependencies in sequential programs are implicit, based on complex *program slicing* [39] that has to account for issues of control flow and mutable state that are absent in dataflow models.

In addition to these overarching benefits, database theory provides us with additional technical tools that are relevant to distributed systems. One of particular interest is the ability to use simple checks for *monotonicity*, the property that the CALM Theorem shows to be both necessary and sufficient for consistent results in the absence of coordination [21]. We can exploit this to decouple code freely across multiple machines without thought for ordering, synchronization or coordination. Another feature of interest is the availability of functional dependencies to describe state invariants that ensure safe partitioning (sharding) of code and state. A third tradition is the body of literature on data provenance [7], which allows data dependencies to be analyzed in subtle ways, with applications to distributed systems including use in efficient fault injection [1]. These topics are beyond the scope of this paper.

1.2 Hydroflow and Prior Work

The open-source implementation of Hydroflow [37] provides the concrete setting for our discussion in this paper. As input, the Hydroflow system takes single-node Hydroflow specs embedded in Rust programs, which can use networking components to communicate with each other. Hydroflow provides the libraries, support routines and compilation scaffolding to allow the Rust compiler (which uses LLVM) to emit high-performance code on each individual node. The details of Hydroflow can be found in the online Hydroflow book [23]; we provide a brief overview here.

At a high level, Hydroflow is similar to many dataflow runtimes and languages, ranging from database system internals going back to System R [5], Ingres [16], and later extensible runtimes like Volcano [14]. Modern readers may be more familiar with contemporary data analytics libraries like Spark [44], Timely Dataflow [34] and Pandas [31]. Hydroflow targets somewhat different performance and correctness goals than the prior work:

Machine Model. As a low-level IR, the goal for Hydroflow is to support programs that can be distributed across both cores and machines at any scale from a single box to the globe and beyond [42]. Hydroflow models the behavior of a set of independent communicating agents ("nodes"), each with its own local state and logical clock. Hydroflow assumes only point-to-point communication, with no assumptions of reliability or ordering on channels, nor built-in facilities for broadcast. In practice, a sender can communicate only with a receiver for whom it has an address in its local state. This captures a standard asynchronous model in which messages between correct nodes can be arbitrarily delayed and reordered, and formally all messages are eventually delivered after an infinite amount of time [12], but in practice delays can be managed via timeouts, which can be specified to arrive as external stimuli to the system. The Hydroflow runtime and language make no further assumptions about failures of nodes or message delivery. The runtime offers general MPMD setups where each node can have different programs and data; more uniform setups are possible as well. Hydroflow does assume a globally-defined namespace for network endpoints (e.g. `IP:port` for internet deployments), but it does not assume individual nodes have knowledge of node membership. The runtime assumes no built-in mechanism for shared state across nodes in the language, but shared memory queues are supported transparently as a communication channel when feasible.

A Single-Node Kernel With Networking Support. Many modern dataflow systems from the "Big Data" era are designed for parallel execution across multiple nodes. In the Hydro stack, any cross-thread "global" model—be it for analytics, live services, or other applications—is the purview of the higher-level Hydrologic language, which is compiled down into Hydroflow. Hydroflow itself is a single-threaded language intended to be run on a single core, with communication support (both shared-memory and networking) allowing multiple Hydroflow instances to run in parallel and communicate efficiently. Using a rough analogy to parallel database systems, Hydrologic's global view is akin to SQL, whereas Hydroflow is a "query plan" language and compiler for an individual core participating in the execution of a parallel query.

Low-latency Data Handling. Many Big Data and Warehousing-centric systems focus on throughput and bulk-synchronous processing. While this is possible in Hydro, we also target low-latency performance for handling asynchronous network events. In this sense Hydroflow is closer to the Click router [26] than Big Data systems like Spark. Like Click, Hydroflow includes support for efficiently managing "push" and "pull" dataflow operators, harnessing the Rust compiler's monomorphization techniques to the task of compiling push/pull dataflows into highly-efficient code that is aggressively "inlined" [36].

Algebraic Typing for Distributed Consistency. Hydroflow builds on research in exploiting formal properties like monotonicity [22] for assessing the distributed consistency properties of programs. Like Bloom [2], it provides a rich dataflow model for composing complex programs, and non-monotonicity analysis to identify program locations that require coordination for consistency. Like LVars [27], Bloom^L, Lasp [32], Gallifrey [33] and Datafun [4], it uses algebraic properties of join semi-lattices—namely Associativity, Commutativity and Idempotence (a.k.a "ACID 2.0" [18])—to distinguish monotonic code fragments from those that require coordination for consistency. Hydroflow is unique in formally modeling the properties of the dataflow runtime itself using join semi-lattices.

Hydroflow's modeling of dataflow as a join semi-lattice drives a number of our optimization examples below, so we discuss it in more detail in the next section.

Before proceeding, we should address common concerns about performance. Empirically, Hydroflow's performance is hitting performance targets we set at the outset of the project. For example, a Hydroflow implementation of Compartmentalized Paxos [40] provides better latency *and* peak throughput than the original handwritten Scala code that was state-of-the-art two years ago [20]. Similarly, a Hydroflow implementation of the Anna key-value store outperforms the original handwritten C++ code and matches its linear scaling under conflict [20]; the original Anna paper was already providing performance under contention that was orders of magnitude faster than research and production systems like Redis and Masstree [42]. Raw performance is no longer one of our primary concerns; optimization is the next challenge.

2 DATAFLOW, NETWORKS, AND LATTICES

Dataflow is a widely-used programming model for composing simple data operators into complex programs represented as directed

```
source_stream(shopping) -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join()
  -> map(|(client, (li, class))| ((client, class), li))
  -> group_by(Vec::new, Vec::push)
  -> map(|m| (m, out_addr)) -> dest_sink_serde(out);
```

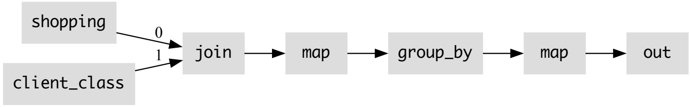

Figure 2: Original Flow

graphs. In many dataflow systems, the operators have formal semantics, often as a superset of the relational algebra. By contrast, the semantics of “edges” in the dataflow are often implicit in an execution model. Typically, the implicit assumption is that an edge from a producer operator `P` to consumer operator `Q` (denoted `P -> Q`) indicates that `P` delivers a *stream* of data to `Q`. This stream semantics implies an ordering constraint: if `P` delivers data in a certain order, `Q` will receive it in that order. In Section 4.1 we discuss how this can be formalized, but for now we can imagine that the producer implicitly assigns a sequence number to each item it sends, and the consumer is guaranteed to receive those items in monotonically increasing order of those sequence numbers.

Dataflow operators can also pass data over a network, a special kind of dataflow edge. A “network edge” of this type *downgrades* the type of communication from ordered streams, by garbling the ordering, batching, and number of transmissions of each item in the stream. If we again imagine implicit sequence numbers at the producer, the consumer has no guarantee of receiving data over a network edge by monotonically increasing position[1]. In the absence of these guarantees, it is difficult to reason about the consistency of program outcomes across multiple executions. In particular, if a program with network edges is replicated, the replicas may be inconsistent; alternatively, if such a program crashes and is re-executed (e.g. by a recovery protocol) the second run may not match the prefix of outcomes that happened in the initial run.

To address these concerns, various projects across decades of work have explored the idea that *certain operators remain consistent even when run on networked edges and/or replicated.* If the operators are inherently Associative, Commutative and Idempotent in their handling of input data, they *upgrade* the dataflow back to consistency of outcomes across networked executions (see [4, 13, 18, 27, 32, 33, 38], etc.) Mathematically, an operator with these properties is a *join semi-lattice* [38] (henceforth we will just use the term “lattice”), and monotonic. These monotonic lattice operators will produce identical outcomes in the face of data arriving in different batches (associativity), orders (commutativity) or multiplicity (idempotence). These properties are compositional, so a dataflow composed of join semi-lattices is itself a composite join semi-lattice. If we can transform our code—whether it be the entire program, or just a component—to use lattices exclusively, we can rest easy about the distinctions between local and networked edges within that scope. The data types will ensure deterministic outcomes across networks[2].

Given this background, one of the goals of optimization in Hydro is to transform programs to make liberal use of lattice operators. To ensure consistency—-that is, determinism across runs and replicas—code segments using non-lattice operators either must run on a single sequential core, or, if distributed, must establish consensus on the order of operations using a protocol like Paxos [29], which often negatively affects latency and availability [22].

Because lattices lie at the heart of our goals of correctly auto-distributing programs, we focus on foundational lattice-oriented transformations in this paper. There are of course many more transformations that are relevant to distributed program optimization. Two that we have explored extensively in implementing Paxos over Hydroflow are auto-decoupling of subprograms and auto-partitioning (sharding) of code and state [20]. However here we stay focused on initial optimizations that demonstrate our ability to safely distribute a simple example in multiple ways.

3 A CLASSIC SCENARIO: SHOPPING CARTS

To illustrate the potential for Hydrolysis, we show how a compiler can take a simple single-node Hydroflow program and transform it step-by-step into a semantically equivalent distributed alternative with a clever twist from the literature. Specifically, we consider the classic problem of implementing a shopping cart, inspired by the Amazon Dynamo paper [11]. Our goal will be to step through a sequence of individual transformations in the Hydrolysis search space, as an example of the transformation paths that Hydrolysis would enumerate. All the code we show below runs correctly in Hydroflow, and is available in full at https://github.com/hydro-project/hydroflow/tree/applied23/hydroflow/examples/shopping.

Given a single-node implementation of a shopping cart system, we partition the program between client and server, replicate the servers for fault tolerance, and introduce an optimization from Conway et al. [10] to work with lattices throughout—hence allowing not only shopping but also checkout to proceed without using any distributed coordination.

Our naive Hydroflow code for the shopping carts is shown in Figure 2, along with an auto-generated dataflow diagram of the code. We envision two classes of shopping data, one for basic customers and one for “premium” customers. The type of the shopping

[1] Some networking protocols like TCP offer reliable ordered delivery. These protocols are not a panacaea for many applications, however—it’s quite common for TCP sessions to terminate unpredictably. As a result, many long-running services layer their own solutions to ordering and reliability on top of multiple unreliable TCP sessions [17].

[2] The CALM Theorem [22] proves that this relationship is bidirectional: programs are consistent across unreliable network edges if *and only if* they provide monotonicity guarantees of the form guaranteed by lattices. CALM was proven in a formal framework of distributed logic programming rather than distributed algebra, so there are still some technical details to be done to apply this argument to a language like Hydroflow, but the intuition is fairly direct.

data streaming in is of the form `Stream<ClientLineItem>`—an unbounded list of requests. `ClientLineItem` is a nested pair `(client: usize, (item: String, qty: i16))` representing a request from a client with a non-negative integer ID (Rust's `usize` type) to add a quantity of a specific item to their shopping cart, or delete a quantity of an item (via negative `qty`). In this simple program, a shopping cart is similar to string data, but rather than being an ordered list of characters, it is an ordered list of `ClientLineItems`[3].

We begin in Figure 2 with a single-node Hydroflow implementation that we envision being generated naively from a distribution-agnostic Hydrologic spec. We will walk through this code in some detail; subsequent snippets in the paper use substantially the same operators, just reconfigured via various transformations.

In the Hydroflow language, `->` represents a stream of data flowing from a producing *operator* to a consuming *operator* on a single node. Hydroflow offers a variety of operators familiar from relational algebra and functional languages like Spark or Pandas, including the ability to embed "user-defined functions" (i.e. arbitrary single-node sequential code) in operators like `map` and `reduce`.

Taking the code one line at a time, we begin in Line 1 with a `source_stream` operator that takes an unbounded stream of packets as they arrive from an IP port (defined in a variable `shopping` in a Rust prelude to the Hydroflow program) and passes them to the first input (input `[0]`) of a subgraph called `lookup_class`. Line 2 begins with a `source_iter` operator that iterates once through a iterable Rust collection (defined in the variable `client_class`) in a Rust prelude to the program) and passes the results to the second input (input `[1]`) of the `lookup_class` subgraph.

The remaining four lines specify `lookup_class`. Line 3 specifies a relational equijoin on `(key, value)` pairs from the two inputs; inputs that match by key are concatenated in an output tuple of the form `(key, (value0, value1))`. Line 4 is a `map` function that takes the output of the join and reformats it to suit the next operator in Line 5. This is a SQL-style `group_by` that accepts tuples of the form `(key, value)`—in this case the preceding `map` generates a key of the form `(client, class)` and a value `li`.The `group_by` partitions the data by key, and per key it aggregates ("folds") the values using a pair of an initialization function (in this case a Rust `Vec::new` declaring an empty vector) and an iteration function (in this case `Vec::push` which pushes each tuple to the end of the vector). The result is a stream of tuples, one per distinct key. In Line 6 we have two operators that together do network transmission. The first is a `map` function to format tuples of the form `(payload, destination)`, and the second a `dest_sink_serde` that serializes each payload `m` via internal Rust libraries and ships each one to the Hydroflow node at the destination `out_addr` (a Rust variable defined in the prelude).

More intuitively, this flow iterates through customer requests via the `source_stream` operator. It then does a join with a stored `client_class` table to look up a unique `ClientClass` tag for each client via the `join` operator, and loads the tagged shopping requests into the stateful `group_by` operator. The `group_by` operator is initialized with an empty vector (generated by `Vec::new`) which it accumulates by pushing each `LineItem` that arrives to the end of the list. The result is tagged via `map` with a (externally-provided) destination address `out_addr` and sent over the network in serialized form by `dest_sink_serde`.

The shopping stream grows monotonically *without bound*. This means that the `group_by` operator is never able to assemble a "final" answer for a group; even if we hacked it to "time out", at best it could output a "string prefix" (lower bound) of future answers. If we were to allow the `group_by` to pass values to the network, then the external agent at `out_addr` could see non-deterministic prefixes of the shopping cart. If we replicated this code to another node in the system, it might make different choices about which `group_by` values to send out on the network, leading to inconsistency.

For correctness, then, the `group_by` in this case can only output final results for each client "at the end of time"—i.e. when some operational semantics of the system determines that the stream flowing into the `group_by` will never produce more data. While we could augment our program with another channel for client "checkout" messages, that would still not help our dataflow system understand the application semantics of when to release the `group_by` data, because checkouts could race with orders! Instead, we'd like to capture "end-of-stream" explicitly in our type system so the developer can inform the `group_by` operator how to reason formally about safe release of outputs. We address this issue in the next section.

4 TYPE UPGRADES FOR SHOPPING CARTS

In this paper, we consider two kinds of transformations: *type upgrades* (this section) and *local graph transformations* (next section).

The *type upgrades* we seek are all *bounded join semi-lattice* types. As discussed in Section 2, if we can convert to lattice types for our operators, we can override the problems introduced by networks. But we want more than just join semi-lattices; we want *bounded join semi-lattices* (henceforth "bounded lattices") that have a well-defined finite "top" element $\top$. This element at the top of the lattice has the property that once the operator's output reaches $\top$, it will remain $\top$ in the face of any new input. This allows the operator to output the value $\top$ at any time, without fear of future retraction[4].

We now proceed to show how Hydrolysis might transform our program to "upgrade" the types to bounded lattices.

4.1 Bounded Prefix Lattice (BP)

In this variation, the type of the incoming data is a stream of bounded lattice points `Stream<BoundedPrefixLattice`$_S$`>`. The lattice `BoundedPrefixLattice`$_S$ is defined as follows. Given some

[3] It is tempting to assume that shopping requests would be better represented as a set than a stream. The problem is that the quantity of each item needs to be handled carefully. Imagine that a customer orders 2 apples, then orders 2 more apples, then deletes 4 apples. In the end they truly want 0 apples. Two problems arise. One is that sets are idempotent, but counting/summing is not. So the following two sets are equivalent $\{(apple \times 2), (apple \times 2)\} = \{(apple \times 2)\}$, but the following streams are not equivalent $[(apple \times 2), (apple \times 2)] \neq [(apple \times 2)]$. The second problem is that the semantics of deletion and insertion may not be commutative: in some applications, we may ignore "overdrafts" that go below 0. For example, in some definitions, $[(apple \times -4)], [(apple \times 4)] = [(apple \times 4)]$ because the deletion on an empty cart is ignored. Stream semantics ensure these issues are unambiguous.

[4] Bounded semi-lattices may be the only reasonable data types to transmit across a network. Indeed, many consistency tricks in the distributed systems literature attach lattice metadata to objects in the dataflow; TCP sequence numbers, Lamport clocks and vector clocks are three common examples. In effect this metadata "upgrades" the network to use lattices, but this is not typically reflected in a type system for a compiler or debugger to reason about!

```
1 source_stream(shopping_bp) -> [0]lookup_class;
2 source_iter(client_class) -> [1]lookup_class;
3 lookup_class = join()
4   -> map(|(client, (li, class))| ((client, class), li))
5   -> group_by(bp_bot, bp_merge)
6   -> map(|m| (m, out_addr)) -> dest_sink_serde(out);
```

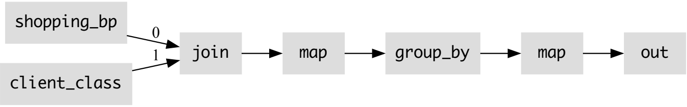

Figure 3: Bounded Prefix Lattice.

fixed-length string S, the domain of the lattice is a set of pairs $(s_i, \texttt{len}(S))$ containing the unique prefix of S of length i, and the full length of S. The semijoin operator $\sqcup$ of this lattice takes two prefixes and simply returns the one with the longer prefix. Note that any two elements of $\texttt{BoundedPrefixLattice}_S$ are guaranteed to share the shorter prefix by definition of both being prefixes of the same S. Similarly, elements of $\texttt{BoundedPrefixLattice}_S$ and $\texttt{BoundedPrefixLattice}_T$ are from different lattices and are incomparable. Each BP has a finite top element $\top = (S, \texttt{len}(S))$, which is identifiable in isolation—it is the only legal element whose first component S has length matching the second component!

We can rewrite our program of Figure 2 to use a BP without changing its output; the result is shown in Figure 3, with the modified shopping_bp input converting each request to a BP type corresponding to its specific shopping session, and tagged with the length of the session (i.e. the number of requests in the session). Rather than streaming individual lineitems (corresponding to "characters" of a "string"), it streams vector prefixes of monotonically increasing length. Notice how the group_by operator is now initialized with the "bottom" ($\perp$) of the lattice via bp_bot, and uses the lattice merge operator bp_merge (line 5) rather than the ad-hoc vector push logic of Figure 2 (line 5). Needless to say using BPs is less efficient in time and space than our original program. But this variation allows us to produce output in bounded time, and the flow is now based entirely on monotonic lattice operators, paving the way for tolerating network edges in subsequent transformations.

4.1.1 *Optimizations on BPs.*

If we know that an edge in our dataflow is a local edge, then we know it preserves ordering and exactly-once delivery between producer and consumer. We can exploit that ordering semantics to implement the BP in a more efficient manner. The result will be similar to our original program based on Vecs, but with guarantees to allow outputs as soon as possible. Specifically, assume we have a producing operator P and consuming operator Q, and P emits a stream of monotonically increasing BP points (i.e. Vec prefixes) across a standard (non-network) edge. We can rewrite the flow segement `P -> Q` as `P -> odiff -> append(len(`S`)) -> Q` where the output of `odiff(`s_j`)` is the "ordered diff"—the suffix of items from the input that were not produced in any previous output $s_i, i < j$—and `append(len(`S`))` maintains a buffer of length `len(`S`)` to reassemble the ordered diffs back into longer and longer prefixes.

This rewrite preserves the BP semantics for P and Q while avoiding the space consumption and data copying of redundant prefixes. Stream termination is detected when the append buffer is full. For correctness, this optimization *requires* that the edge between odiff and append maintains ordering and exactly-once delivery. Ordering comes "for free" on local edges running on a single thread with P and Q. Note that the append operator is not a lattice operator: it is neither associative, nor commutative, nor idempotent. Instead it relies upon the edge itself to be "upgraded" to an ordered, exactly-once delivery.

This optimization opens up the possibility of more optimizations to avoid or postpone reassembling the prefixes. For example, suppose that Q is the operator `map(|s| uppercase(s))`. Then Q runs correctly on odiffs, and hence we can rewrite our program by "pushing" Q earlier in the stream (P `-> odiff ->` Q `-> append`) without changing semantics. This now requires that *all* edges between odiff and append must be ordered and exactly-once, but it ensures that uppercasing is only done once per character. In the most felicitous case, we are able to optimize a single-node program by "pushing" the odiff operator to the beginning of the flow, and the append operator to the end of the flow. In such a case, the flow becomes an intuitive local, ordered stream of small individual items. As a separate optimization, we may be able to fuse odiff into P, or append into Q in certain circumstances. In our original program, the 'group_by' was doing precisely the work of append, so if it was next to an append operator we could entirely delete the append without changing semantics.

These optimizations do not always apply. Moreover, optimized BPs require local edges. As an alternative, we shift attention to an alternate (isomorphic!) structure, the Sealed Set of Indexed Values, a fully lattice-based approach that works across "downgraded" network edges with relatively small space overheads.

4.2 Sealed Set of Indexed Values Lattice (SSIV)

The idea with the SSIV is to embrace the idea of "diffs", but allow them to be accumulated in an ACI fashion. Borrowing ideas from TCP and Conway, et al. [10], we exploit two tricks simultaneously to get a bounded lattice. To begin, we can represent a string S as a set of indexed values `(value, pos)` accumulated via union (sets with union form a lattice!), where pos is a natural number representing a position (index) in the string. Having converted from a vector to a set lattice, we can use a simple trick to bound the lattice. Specifically, a producer can count the size of the set (the length of the string) while enumerating, and piggyback the size on the last element it produces to form a bound or "seal". Once a consumer knows the set size and has received that many distinct elements (possibly out of order!), it knows locally—without any coordination—that it has reached the top of the lattice $\top$ and no new information will be forthcoming. Physically, our representation of items in a sealed set is a triple `(pos, val, Option<len(`S`)>)`, where pos is an index between 0 and `len(`S`) - 1`, val is the value in position pos, and

```
source_stream(shopping_ssiv) -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join()
  -> map(|(client, (li, class))| ((client, class), li))
  -> group_by(ssiv_bot, ssiv_merge)
  -> map(|m| (m, out_addr)) -> dest_sink_serde(out);
```

Figure 4: Sealed Set of Indexed Values. Dataflow diagram is identical to Figure 3, except `ssiv` replaces `bp`.

`len`(S) is an optional field—if provided, it is the length of the string. In Figure 4 we reconsider our example, using a SSIV. The code is identical to that of Figure 3, but using SSIVs instead of BPs.

5 LOCAL GRAPH TRANSFORMATIONS

In this section we examing some dataflow graph transformations that, in concert with our lattice-typed shopping carts, allow us to deliver a fully monotonic, lattice-based implementation of our program. This in turn enables graph transformations for safely distributing the program without any coordination.

5.1 Push Group By Through Join

In Figures 3 and 4, we have a chain of operators `join -> map -> group_by`, where the join finds matches based on `client`, the map simply rearranges the data into `(key, val)` pairs to conform to the `join` API, and the `group_by` partitions on the pair `(client, class)`. As mentioned previously, there is one unique `class` per `client`; that is, we have a functional dependency `client` → `class`. That means that the groups are partitioned uniquely by `client` alone; the `class` is simply a deterministic function of the `client`. This presents an opportunity for a classic query optimization that pushes the `group_by` through the join (e.g. [6, 43]). The resulting program is shown in Figure 5.

This transformation can improve performance significantly, because we now look up the client's `ClientClass` once per *sealed cart* (after the `group_by`) rather than once per lineitem request. This was the intended goal of this optimization in the early literature. Perhaps more interesting for this paper, pushing the `group_by` down before the join means that lineitems need not be stored on the same node as the `client_class` table, as we will discuss next.

5.2 Decoupling Across a Network

Thanks to the type upgrades of Section 4, our dataflow is now fully composed of monotonic lattice operators[5]. Note that lattices are used throughout the entire shopping cart lifecycle: not just for cart add and delete requests, but also for checkout, which is a monotone "threshold test" for ⊤ on the bounded semi-lattice of a session. This "monotone checkout" trick is the one we borrow from Conway, et al. [10]. As a result of complete monotonicity, we can use network edges between any of our operators—say separating "client" and "server" components—and count on the operators to provide consistent behavior due to their ACI properties.

In particular, notice that the `group_by` operator maintains the state for each shopping cart. Having pushed the `group_by` down close to the source in our previous transformation, we can now choose to "cut the flow" by introducing a network edge in one of two places. The first option is to put the network edges *upstream* of the `group_by`, as shown in Figure 6. This means that clients are stateless and simply send lineitem requests to the server, which holds the cart state. This design may be useful for fault tolerance, as the server may be replicated (Section 5.3) and hence be more reliable than the client. The second option is that we can put the network edge downstream of the `group_by` as in Figure 7. This results in the cart state being accumulated on the client side of the network, which may be favorable for concerns of governance or privacy. Note that the `client_class` table mixes vendor-centric information about many clients, and seems reasonable to store at the server, but the transient state of the cart is kept at the client. Hence the server only sees carts after checkout; if a client regrets adding something to their cart and subsequently deletes it before checkout, only the client will know that. This optimization choice reflects a nuanced data ownership position that sits between a fully stateless client implementation, and a *local first* [25] design in which no state is stored on servers. Other choices for state partitioning are possible as well via related transformation choices.

5.3 Server Replication

Another advantage of our type "upgrade" to lattices is that we can replicate our stateful component for fault tolerance and/or geo-locality, and have the various replicas broadcast updates amongst each other to reach eventual consistency, outputting results whenever all information becomes available (⊤). In Figure 8 we show a replicated version of the stateful server in Figure 6. Note that the figure omits the unmodified "client" logic from line 1 of Figure 6 to keep the dataflow diagram visible.

Getting from the singleton server code to the replicated server code requires the application of a number of transformations. Space prevents us from stepping through them one by one. In brief, the transformation flow is as follows:

(1) We add logic to `tee` the shopping carts to a "broadcast" channel (lines 4-6 of Figure 8).
(2) We add logic to send the broadcast to all server addresses; each message is replicated for each server via a cartesian product operator (aka `cross_join`), and then sent to each server (line 6-8).
(3) We add logic to receive the broadcast and merge the remote updates into the local flow via an additional copy of the (idempotent!) lattice merge via `group_by` (lines 9-10) and avoid sending redundant updates via `unique`[6].

[5] The only potential non-monotonic operator in our example was the `group_by`. Relational join is a monotonic lattice *morphism* over the cross-product domain of its inputs, and purely functional `map` functions are lattice morphisms as well with respect to the set of items passed into them [10].

[6] Some readers may note that the first `group_by` operator is now unnecessary for correctness; it offers a sub-aggregation, but the second `group_by` could instead aggregate individual lineitem requests from clients as well as broadcasts from replicas. It is a matter of optimization whether the earlier `group_by` should be elided; this again fits in the realm of classical query optimization, and would be explored by Hydrolysis in a full implementation.

```
source_stream(shopping_ssiv)
  -> group_by(ssiv_bot, ssiv_merge)
  -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join()
  -> map(|(client, (li, class))| ((client, class), li))
  -> map(|m| (m, out_addr))
  -> dest_sink_serde(out);
```

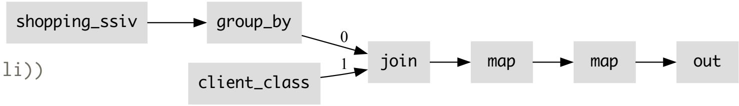

Figure 5: Push Group By Through Join

```
source_stream(shopping_ssiv) -> map(|pair| (pair, addr1))  -> dest_sink_serde(reqs_out);
source_stream_serde(reqs_in) -> map(|((client, req), _a): ((usize, ReqSsivLattice), _)| (client, req))
  -> group_by(ssiv_bot, ssiv_merge) -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join() -> map(|(client, (li, class))| ((client, class), li)) -> map(|m| (m, out_addr))
  -> dest_sink_serde(out);
```

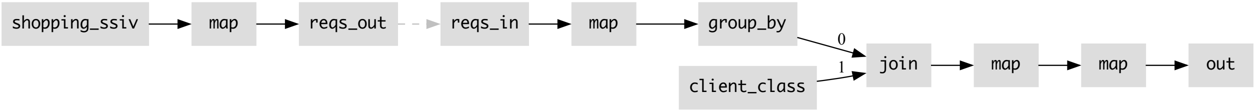

Figure 6: Decouple Across a Network: Server-Side Cart State

```
source_stream(shopping_ssiv) -> group_by(ssiv_bot, ssiv_merge) -> map(|pair| (pair, addr1)) -> dest_sink_serde(basic_out);
source_stream_serde(basic_in) -> map(|((client, cart), _a): ((usize, ReqSsivLattice), _)| (client, cart))
  -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join() -> map(|(client, (li, class))| ((client, class), li)) -> map(|m| (m, out_addr)) -> dest_sink_serde(out);
```

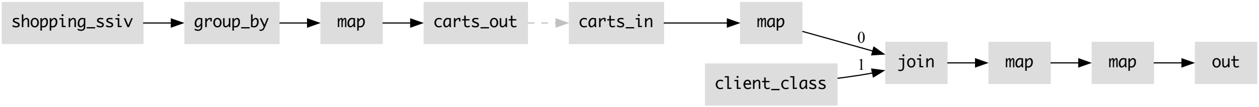

Figure 7: Decouple Across a Network: Client-Side State

6 DISCUSSION AND FUTURE WORK

This paper captures a snapshot of our early explorations of the potential of dataflow optimization as a vehicle for optimizing distributed programs. We are increasingly optimistic that a dataflow kernel like Hydroflow is a useful optimization target for distributed systems concerns. By incorporating the ACI properties of lattices into our type system we can reason about allowing network communication to be introduced safely into programs. As illustrated in Section 4 we are beginning to see the potential for an optimizer to automatically "upgrade" programs to use lattice types without changing semantics. This in turn opens up opportunities for decoupling and replication of program components across networks.

In Section 5.2 we saw that two different choices of program transformations can address different objectives: in that case a tradeoff between fault tolerance on one hand, and governance/privacy on another. This suggests that the objective function and constraints for optimizing modern distributed programs may be quite a bit more varied and nuanced than classical query compilation.

This workshop paper is early, and we are eagerly pursuing a number of directions to go from these concepts and hand-optimizations

```
source_stream_serde(reqs_in) -> map(|((client, cart), _a): ((usize, ReqSsivLattice), _)| (client, cart))
    -> group_by(ssiv_bot, ssiv_merge) -> [0]lookup_class;
source_iter(client_class) -> [1]lookup_class;
lookup_class = join() -> map(|(client, (li, class))| ((client, class), li) ) -> tee();
lookup_class[clients] -> all_in;
lookup_class[broadcast] -> [0]broadcast;
source_stream(server_addrs) -> [1]broadcast;
broadcast = cross_join() -> dest_sink_serde(broadcast_out);
source_stream_serde(broadcast_in) -> map(|(m, _a): (((usize, ClientClass), ReqSsivLattice), _)| m) -> all_in;
all_in = merge() -> group_by(ssiv_bot, ssiv_merge) -> unique()
  -> map(|m| (m, out_addr)) -> dest_sink_serde(out);

```

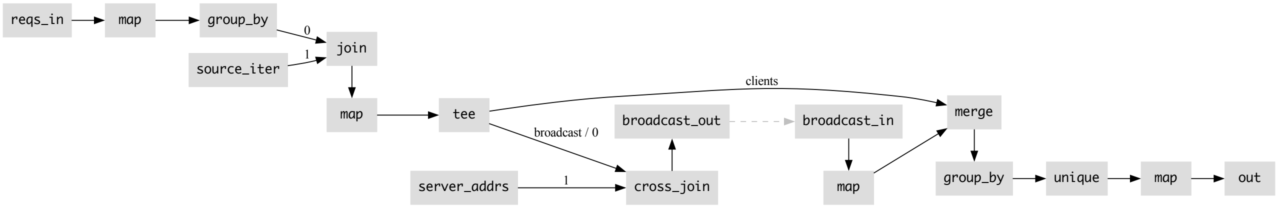

Figure 8: Replicated Server with Broadcast.

to a rich, automated reality. The agenda encompasses a range of challenges, including the following.

(1) We need a language for our own use to formalize our rewrite rules and prove equivalence of rewritten program fragments.
(2) We need to register a large number of transformation rules. This likely needs to include many classical examples from relational database query optimization, functional programming and stream query processing. In addition, we expect to trip across new optimizations that address issues with cloud deployments.
(3) We need a way for programmers to define multiple objectives they want to optimize, and to express constraints on the optimization space, e.g. for fault tolerance or governance. In the vision paper for Hydro [8] we highlight fault tolerance as a programming "aspect" that developers should be able to specify independent of their program's intended semantics. That vision requires further work, but some seeds are apparent even in our simple example here—namely the ability to consistently replicate components. Simultaneously maintaining fault tolerance constraints and performance objectives is an interesting challenge for an optimizer like Hydrolysis.
(4) We need a transformation-based optimizer that can ingest our rules, objective functions and constraints, and efficiently search the space of equivalent programs to minimize the objective function. We are enthusiastic that recent work on e-graphs could offer an efficient vehicle for our work.

We are optimistic that open-source tools like Egg [41] can make it possible for us to address these challenges relatively quickly. The Hydro stack itself is also open source, and we welcome additional research and development efforts!

ACKNOWLEDGMENTS

This work was supported by unrestricted gifts from Amazon Web Services, Ant Group, Ericsson, Futurewei, Google, Intel, Meta, Microsoft, Scotiabank, VMware, Meta, Astronomer, IBM, Intel, Lacework, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber and NSF CISE Expeditions Award CCF-1730628. Hellerstein's work was done while on partial leave at Sutter Hill Ventures. This work arose from many discussions with Hydro teammates Tiemo Bang, Alvin Cheung, David Chu, Natacha Crooks, Chris Douglas, Justin Jaffray, Lucky Katahanas, Chris Liu, Rithvik Panchapakesan, and Kaushik Shivakumar.

REFERENCES

[1] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. 2017. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Trans. Database Syst.* 42, 4, Article 23 (Oct. 2017), 31 pages. https://doi.org/10.1145/3110214
[2] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach.. In *CIDR*. Citeseer, 249–260.
[3] Peter Alvaro, William R Marczak, Neil Conway, Joseph M Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in time and space. In *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Springer, 262–281.
[4] Michael Arntzenius and Neelakantan R Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 214–227.
[5] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1, 2 (1976), 97–137.
[6] Surajit Chaudhuri and Kyuseok Shim. 1994. Including group-by in query optimization. In *VLDB*, Vol. 94. 12–15.
[7] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.
[8] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Matthew Milano. 2021. New directions in cloud programming. In *Conference on Innovative Data Research (CIDR)*.

[9] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[10] Neil Conway, William R Marczak, Peter Alvaro, Joseph M Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–14.

[11] Giuseppe DeCandia et al. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[13] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *ACM Sigmod Record* 16, 3 (1987), 249–259.

[14] Goetz Graefe. 1994. Volcano: an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

[15] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[16] GD Held, MR Stonebraker, and Eugene Wong. 1975. INGRES: A relational data base system. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*. 409–416.

[17] Pat Helland. 2012. Idempotence is not a medical condition. *Commun. ACM* 55, 5 (2012), 56–65.

[18] Pat Helland and David Campbell. 2009. Building on quicksand. *arXiv preprint arXiv:0909.1788* (2009).

[19] Joseph M. Hellerstein. 2008. The Data-Centric Gambit. *Computing Community Consortium (CCC) Blog* (20 Oct. 2008). https://cccblog.org/2008/10/20/the-data-centric-gambit/

[20] Joseph M. Hellerstein. 2023. Hydroflow Performance Update. (9 May 2023). https://databeta.wordpress.com/2023/05/09/hydroflow-performance-update-whoosh/

[21] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM: When Distributed Consistency is Easy. *Commun. ACM* 63, 9 (Aug. 2020), 72–81. https://doi.org/10.1145/3369736

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

[23] Joseph M. Hellerstein, Lucky Katahanas, and Mingwei Samuel. 2023. The Hydroflow Book. https://hydro-project.github.io/hydroflow/book/, Last accessed on 2023-04-03.

[24] Martin Kleppmann and Alastair R Beresford. 2018. Automerge: Real-time data sync between edge devices. In *1st UK Mobile, Wearable and Ubiquitous Systems Research Symposium (MobiUK 2018). https://mobiuk. org/abstract/S4-P5-Kleppmann-Automerge. pdf*. 101–105.

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

[26] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[27] Lindsey Kuper and Ryan R Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. 71–84.

[28] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M Hellerstein. 2022. Katara: synthesizing CRDTs with verified lifting. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1349–1377.

[29] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, 75–86.

[31] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

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

[33] Matthew Milano, Rolph Recto, Tom Magrino, and Andrew C Myers. 2019. A tour of gallifrey, a language for geodistributed programming. In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

[35] Greg Nelson and Derek C Oppen. 1980. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364.

[36] Mingwei Samuel. 2021. *Hydroflow: A Model and Runtime for Distributed Systems Programming*. Master's thesis. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-201.html

[37] Mingwei Samuel, Justin Jaffray, Shadaj Laddad, Joe Hellerstein, Lucky Katahanas, Tyler Hou, Alex Rasmussen, David Chu, Conor Power, Amrita Rajan, and Rithvik Panchapakesan. 2023. *Hydroflow*. https://github.com/hydro-project/hydroflow/

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

[39] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[40] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (July 2021), 2203–2215. https://doi.org/10.14778/3476249.3476273

[41] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021).

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

[43] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *Proceedings of 21th International Conference on Very Large Data Bases*. 345–357.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.