Optimizing Distributed Protocols with Query Rewrites

DAVID C. Y. CHU, University of California, Berkeley, USA

RITHVIK PANCHAPAKESAN, University of California, Berkeley, USA

SHADAJ LADDAD, University of California, Berkeley, USA

LUCKY E. KATAHANAS, Sutter Hill Ventures, USA

CHRIS LIU, University of California, Berkeley, USA

KAUSHIK SHIVAKUMAR, University of California, Berkeley, USA

NATACHA CROOKS, University of California, Berkeley, USA

JOSEPH M. HELLERSTEIN, University of California, Berkeley, USA and Sutter Hill Ventures, USA

HEIDI HOWARD, Azure Research, Microsoft, UK

Distributed protocols such as 2PC and Paxos lie at the core of many systems in the cloud, but standard implementations do not scale. New scalable distributed protocols are developed through careful analysis and rewrites, but this process is ad hoc and error-prone. This paper presents an approach for scaling *any* distributed protocol by applying rule-driven rewrites, borrowing from query optimization. Distributed protocol rewrites entail a new burden: reasoning about spatiotemporal correctness. We leverage order-insensitivity and data dependency analysis to systematically identify correct coordination-free scaling opportunities. We apply this analysis to create preconditions and mechanisms for coordination-free decoupling and partitioning, two fundamental vertical and horizontal scaling techniques. Manual rule-driven applications of decoupling and partitioning improve the throughput of 2PC by 5× and Paxos by 3×, and match state-of-the-art throughput in recent work. These results point the way toward automated optimizers for distributed protocols based on correct-by-construction rewrite rules.

CCS Concepts: • **Computing methodologies → Distributed computing methodologies**; • **Information systems → Query optimization**.

Additional Key Words and Phrases: Distributed Systems, Query Optimization, Paxos, 2PC, Relational Algebra, Datalog, Partitioning, Dataflow, Monotonicity

Authors' addresses: David C. Y. Chu, University of California, Berkeley, USA, thedavidchu@berkeley.edu; Rithvik Panchapakesan, University of California, Berkeley, USA, rithvik@berkeley.edu; Shadaj Laddad, University of California, Berkeley, USA, shadaj@berkeley.edu; Lucky E. Katahanas, Sutter Hill Ventures, USA, lucky@shv.com; Chris Liu, University of California, Berkeley, USA, chris-liu@berkeley.edu; Kaushik Shivakumar, University of California, Berkeley, USA, kaushiks@berkeley.edu; Natacha Crooks, University of California, Berkeley, USA, ncrooks@berkeley.edu; Joseph M. Hellerstein, University of California, Berkeley, USA and Sutter Hill Ventures, USA, hellerstein@berkeley.edu; Heidi Howard, Azure Research, Microsoft, UK, heidi.howard@microsoft.com.

ACM 2836-6573/2024/2-ART2
https://doi.org/10.1145/3639257

ACM Reference Format:

David C. Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites. *Proc. ACM Manag. Data* 2, N1 (SIGMOD), Article 2 (February 2024), 25 pages. https://doi.org/10.1145/3639257

1 INTRODUCTION

Promises of better cost and scalability have driven the migration of database systems to the cloud. Yet, the distributed protocols at the core of these systems, such as 2PC [46] or Paxos [43], are not designed to scale: when the number of machines grows, overheads often increase and throughput drops. As such, there has been a wealth of research on developing new, scalable distributed protocols. Unfortunately, each new design requires careful examination of prior work and new correctness proofs; the process is ad hoc and often error-prone [2, 35, 51, 53, 57, 62]. Moreover, due to the heterogeneity of proposed approaches, each new insight is localized to its particular protocol and cannot easily be composed with other efforts.

This paper offers an alternative approach. Instead of creating new distributed protocols from scratch, we formalize scalability optimizations into *rule-driven rewrites* that are correct by construction and can be applied to *any* distributed protocol.

To rewrite distributed protocols, we take a page from traditional SQL query optimizations. Prior work has shown that distributed protocols can be expressed declaratively as sets of queries in a SQL-like language such as Dedalus [6], which we adopt here. Applying query optimization to these protocols thus seems like an appealing way forward. Doing so correctly however, requires care, as the domain of distributed protocols requires optimizer transformations whose correctness is subtler than classical matters like the associativity and commutativity of join. In particular, transformations to scale across machines must reason about program equivalence in the face of changes to spatiotemporal semantics like the order of data arrivals and the location of state.

We focus on applying two fundamental scaling optimizations in this paper: *decoupling* and *partitioning*, which correspond to vertical and horizontal scaling. We target these two techniques because (1) they can be generalized across protocols and (2) were recently shown by Whittaker et al. [63] to achieve state-of-the-art throughput on complex distributed protocols such as Paxos. While Whittaker's rewrites are handcrafted specifically for Paxos, our goal is to rigorously define the general preconditions and mechanics for decoupling and partitioning, so they can be used to correctly rewrite *any* distributed protocol.

Decoupling improves scalability by spreading *logic* across machines to take advantage of additional physical resources and pipeline parallel computation. Decoupling rewrites data dependencies on a single node into messages that are sent via asynchronous channels between nodes. Without coordination, the original timing and ordering of messages cannot be guaranteed once these channels are introduced. To preserve correctness without introducing coordination, we decouple sub-components that produce the same responses regardless of message ordering or timing: these sub-components are *order-insensitive*. Order-insensitivity is easy to systematically identify in Dedalus thanks to its relational model: Dedalus programs are an (unordered) set of queries over (unordered) relations, so the logic for ordering—time, causality, log sequence numbers—is the exception, not the norm, and easy to identify. By avoiding decoupling the logic that explicitly relies on order, we can decouple the remaining order-insensitive sub-components without coordination.

Partitioning improves scalability by spreading *state* across machines and parallelizing compute, a technique widely used in query processing [22, 25]. Textbook discussions focus on partitioning data to satisfy a single query operator like join or group-by. If the next operator downstream requires a different partitioning, then data must be forwarded or “shuffled” across the network. We would like to partition data in such a way that *entire sub-programs* can compute on local data without reshuffling. We leverage relational techniques like functional dependency analysis to find data partitioning schemes that can allow as much code as possible to work on local partitions without reshuffling between operators. This is a benefit of choosing to express distributed protocols in the relational model: functional dependencies are far easier to identify in a relational language than a procedural language.

We demonstrate the generality of our optimizations by methodically applying rewrites to three seminal distributed protocols: voting, 2PC, and Paxos. We specifically target Paxos [59] as it is a protocol with many distributed invariants and it is challenging to verify [31, 66, 67]. The throughput of the optimized voting, 2PC, and Paxos protocols scale by 2×, 5×, and 3× respectively, a scale-up factor that matches the performance of ad hoc rewrites [63] when the underlying language of each implementation is accounted for and achieves state-of-the-art performance for Paxos.

Our correctness arguments focus on the equivalence of localized, “peephole” optimizations of dataflow graphs. Traditional protocol optimizations often make wholesale modifications to protocol logic and therefore require holistic reasoning to prove correctness. We take a different approach. Our rewrite rules modify existing programs with small local changes, each of which is proven to preserve semantics. As a result, each rewritten subprogram is provably indistinguishable to an observer (or client) from the original. We do not need to prove that holistic protocol invariants are preserved—they must be. Moreover, because rewrites are local and preserve semantics, they can be *composed* to produce protocols with multiple optimizations, as we demonstrate in Section 5.2.

Our local-first approach naturally has a potential cost: the space of protocol optimization is limited by design as it treats the initial implementation as “law”. It cannot distinguish between true protocol invariants and implementation artifacts, limiting the space of potential optimizations. Nonetheless, we find that, when applying our results to seminal distributed system algorithms, we easily match the results of their (manually proven) optimized implementations.

In summary, we make the following contributions:

(1) We present the preconditions and mechanisms for applying multiple correct-by-construction rewrites of two fundamental transformations: decoupling and partitioning.

(2) We demonstrate the application of these rule-driven rewrites by manually applying them to complex distributed protocols such as Paxos.

(3) We evaluate our optimized programs and observe $2-5\times$ improvement in throughput across protocols with state-of-the-art throughput in Paxos, validating the role of correct-by-construction rewrites for distributed protocols.

Due to a lack of space, the full precondition, mechanism, and proof of correctness of each rewrite in this paper can be found in the technical report [16].

2 BACKGROUND

Our contributions begin with the program rewriting rules in Section 3. Naturally, the correctness of those rules depends on the details of the language we are rewriting, Dedalus. Hence in this section

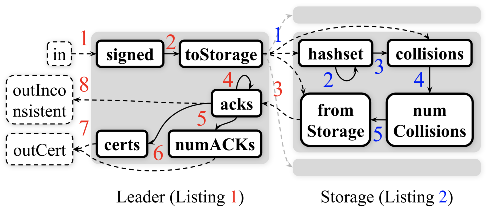


Fig. 1. Dataflow diagram for a verifiably-replicated KVS. Edges are labeled with corresponding line numbers; dashed edges represent asynchronous channels. Each gray bounding box represents a node; select nodes' dataflows are presented.

we pause to review the syntax and semantics of Dedalus, as well as additional terminology we will use in subsequent discussion.

Dedalus is a spatiotemporal logic for distributed systems [6]. As we will see in Section 2.3, Dedalus captures specifications for the state, computation and messages of a set of distributed **nodes** over time. Each node (a.k.a. machine, thread) has its own explicit "clock" that marks out local time sequentially. Dedalus (and hence our work here) assumes a standard asynchronous model in which messages between correct nodes can be arbitrarily delayed and reordered, but must eventually be delivered after an infinite amount of time [24].

Dedalus is a dialect of Datalog¬, which is itself a SQL-like declarative logic language that supports familiar constructs like joins, selection, and projection, with additional support for recursion, aggregation (akin to `GROUP BY` in SQL), and negation (`NOT IN`). Unlike SQL, Datalog¬ has set semantics.

2.1 Running example

As a running example, we focus on a verifiably replicated key-value store with hash-conflict detection inspired by [56]. We use this example to explain the core concepts of Dedalus and to illustrate in Sections 3 and 4 how our transformations can be applied. In Section 5 we turn our attention to more complex and realistic examples, including Paxos and 2PC. Figure 1 provides a high level diagram of the example; we explain the corresponding Dedalus code (Listings 1 and 2) in the next subsection.

The running example consists of a leader node and multiple storage nodes and allows clients to write to storage nodes, with the ability to detect concurrent writes. The leader node cryptographically signs each client message and broadcasts both the message and signature to each storage node. Each storage node then stores the message and the hash of the message in a local table if the signature is valid. The storage nodes also calculate the number of unique existing messages in the table whose hash collides with the hash of the message. The storage nodes then sign the original message and respond to the leader node. Upon collecting a response from each storage node, if the number of hash collisions is consistent across responses, the leader creates a certificate of all the responses and replies to the client. If any two storage nodes report differing numbers of hash collisions, the leader notifies the client of the inconsistency. We use this simple protocol for illustration, and present more complete protocols—2PC and Paxos—in Section 5.

2.2 Datalog¬

We now introduce the necessary Datalog¬ terminology, copying code snippets from Listings 1 and 2 to introduce key concepts.

A Datalog$^\neg$ **program** is a set of **rules** in no particular order. A rule φ is like a view definition in SQL, defining a virtual relation via a query over other relations. A **literal** in a rule is either a relation, a negated relation, or a boolean expression. A rule consists of a deduction operator `:-` defining a single left-hand-side relation (the **head** of the rule) via a list of right-hand-side literals (the **body**).

Consider Line 3 of Listing 2, which computes hash collisions:

```
3  collisions(val2,hashed,l,t) :- toStorage(val1,leaderSig,l,t), hash(val1,hashed),
     hashset(hashed,val2,l,t)
```

In this example, the head literal is `collisions`, and the body literals are `toStorage`, `hash`, and `hashset`. Each body literal can be a (possibly negated) **relation** r consisting of multiple **attributes** A, *or* a boolean expression; the head literal must be a relation. For example, `hashset` is a relation with four attributes representing the hash, message value, location, and time in that order. Each attribute must be bound to a constant or **variable**; attributes in the head literal can also be bound to **aggregation functions**. In the example above, the attribute representing the message value in `hashset` is bound to the variable `val2`. Positive literals in the body of the rule are joined together; negative literals are anti-joined (SQL's `NOT IN`). Attributes bound to the same variable form an equality predicate—in the rule above, the first attribute of `toStorage` must be equal to the first attribute of `hash` since they are both bound to `val1`; this specifies an equijoin of those two relations. Two positive literals in the same body that share no common variables form a cross-product. Multiple rules may have the same head relation; the head relation is defined as the disjunction (SQL `UNION`) of the rule bodies.

Note how library functions like `hash` are simply modeled as infinite relations of the form `(input, output)`. Because these are infinite relations, they can only be used in a rule body if the input variables are bound to another attribute—this corresponds to "lazily evaluating" the function only for that attribute's finite set of values. For example, the relation `hash` contains the fact `(x, y)` if and only if `hash(x)` equals y.

Relations r are populated with **facts** f, which are tuples of values, one for each attribute of r. We will use the syntax $\pi_A(f)$ to project f to the value of attribute A. Relations with facts stored prior to execution are traditionally called *extensional* relations, and the set of extensional relations is called the **EDB**. Derived relations, defined in the heads of rules, are traditionally called *intensional* relations, and the set of them is called the **IDB**. Boolean operators and library functions like `hash` have pre-defined content, hence they are (infinite) EDB relations.

Datalog$^\neg$ also supports negation and aggregations. An example of aggregation is seen in Listing 2 Line 4, which counts the number of hash collisions with the `count` aggregation:

```
4  numCollisions(count<val>,hashed,l,t) :- collisions(val,hashed,l,t)
```

In this syntax, attributes that appear outside of aggregate functions form the `GROUP BY` list; attributes inside the functions are aggregated. In order to compute aggregation in any rule φ, we must first compute the full content of all relations r in the body of φ. Negation works similarly: if we have a literal `!r(x)` in the body, we can only check that `r` is empty after we're sure we have computed the full contents of `r(x)`. We refer the reader to [1, 48] for further reading on aggregation and negation.

2.3 Dedalus

Dedalus programs are legal Datalog$^\neg$ programs, constrained to adhere to three additional rules on the syntax.

(1) Space and Time in Schema: All IDB relations must contain two attributes at their far right: location L and time T. Together, these attributes model *where* and *when* a fact exists in the system. For example, in the rule on Line 3 discussed above, a `toStorage` message m and signature sig that arrives at time t at a node with location $addr$ is represented as a fact `toStorage(`$m, sig, addr, t$`)`.

(2) Matching Space-Time Variables in Body: The location and time attributes in *all* body literals must be bound to the same variables l and t, respectively. This models the physical property that two facts can be joined only if they exist at the same time and location. In Line 3, a `toStorage` fact that appears on node l at time t can only match with `hashset` facts that are also on l at time t.

We model library functions like `hash` as relations that are known (replicated) across all nodes n and unchanging across all timesteps t. Hence we elide L and T from function and expression literals as a matter of syntax sugar, and assume they can join with other literals at all locations and times.

(3) Space and Time Constraints in Head: The location and time variables in the *head* of rules must obey certain syntactic constraints, which ensure that the "derived" locations and times correspond to physical reality. These constraints differ across three types of rules. **Synchronous** ("deductive" [6]) rules are captured by having the same time variable in the head literal as in the body literals. Having these derivations assigned to the same timestep t is only physically possible on a single node, so the location in the head of a synchronous rule must match the body as well. **Sequential** ("inductive" [6]) rules are captured by having the head literal's time be the successor (`t+1`) of the body literals' times `t`. Again, sequentiality can only be guaranteed physically on a single node in an asychronous system, so the location of the head in a sequential rule must match the body. **Asynchronous** rules capture message passing between nodes, by having different time and location variables in the head than the body. In an asynchronous system, messages are delivered at an arbitrary time in the future. We discuss how this is modeled next.

In an asynchronous rule φ, the location attribute of the head and body relations in φ are bound to different variables; a different location in the head of φ indicates the arrival of the fact on a new node. Asynchronous rules are constrained to capture non-deterministic delay by including a body literal for the built-in `delay` relation (a.k.a. `choose` [6], `chosen` [4]), a non-deterministic function that independently maps each head fact to an arrival time. The logical formalism of the `delay` function is discussed in [4]; for our purposes it is sufficient to know that `delay` is constrained to reflect Lamport's "happens-before" relation for each fact. That is, a fact sent at time t on l arrives at time t' on l', where $t < t'$. We focus on Listing 2, Line 5 from our running example.

```
5  fromStorage(l,sig,val,collCnt,l',t') :- toStorage(val,leaderSig,l,t),
     hash(val,hashed), numCollisions(collCnt,hashed,l,t), sign(val,sig),
     leader(l'), delay((sig,val,collCnt,l,t,l'),t')
```

This is an asynchronous rule where a storage node l sends the count of hash collisions for each distinct storage request back to the leader l'. Note the `l'` and `t'` in the head literal: they are derived from the body literals `leader` (an EDB relation storing the leader address) and the built-in `delay`. Note also how the first attribute of `delay` (the function "input") is a tuple of variables that, together, distinguish each individual head fact. This allows `delay` to choose a different `t'` for every head fact [4]. The `l` in the head literal represents the storage node's address and is used by the leader to count the number of votes; it is unrelated to asynchrony.

So far, we have only talked about facts that exist at a point in time t. State change in Dedalus is modeled through the existence or non-existence of facts *across* time. **Persistence rules** like the

Listing 1. Hashset leader in Dedalus.

```
1 signed(val,leaderSig,l,t) :- in(val,l,t), sign(val,leaderSig)
2 toStorage(val,leaderSig,l',t') :- signed(val,leaderSig,l,t), storageNodes(l'),
    delay((val,leaderSig,l,t,l'),t')
3 acks(src,sig,val,collCnt,l,t) :- fromStorage(src,sig,val,collCnt,l,t)
4 acks(src,sig,val,collCnt,l,t') :- acks(src,sig,val,collCnt,l,t), t'=t+1
5 numACKs(count<src>,val,collCnt,l,t) :- acks(src,sig,val,collCnt,l,t)
6 certs(cert<sig>,val,collCnt,l,t) :- acks(src,sig,val,collCnt,l,t)
7 outCert(cer,val,collCnt,hashed,l',t') :- certs(ce,val,collCnt,l,t),
    numACKs(cnt,val,collCnt,l,t), numNodes(cnt), client(l'),
    delay((cer,val,collCnt,hashed,l,t,l'),t')
8 outInconsistent(val,l',t') :- acks(src1,sig1,val,collCnt1,l,t),
    acks(src2,sig2,val,collCnt2,l,t), collCnt1 != collCnt2, client(l'),
    delay((val,l,t,l'),t')
```

one below from Line 2 of Listing 2 ensure, inductively, that facts in `hashset` that exist at time t exist at time $t+1$. Relations with persistence rules—like `hashset`—are **persisted**.

```
2 hashset(hashed,val,l,t') :- hashset(hashed,val,l,t), t'=t+1
```

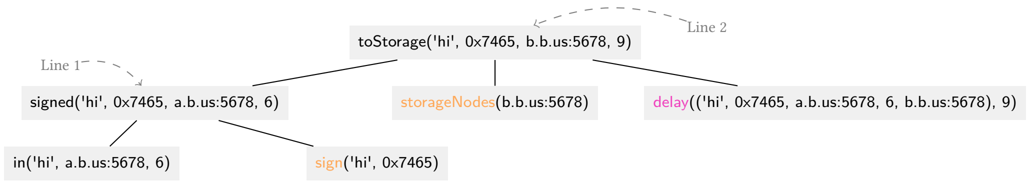
2.4 Further terminology

We introduce some additional terminology to capture the rewrites we wish to perform on Dedalus programs.

We assume that Dedalus programs are composed of separate **components** C, each with a non-empty set of rules $\overline{\varphi}$. In our running example, Listings 1 and 2 define the leader component and the storage component. All the rules of a component are executed together on a single physical node. Many instances of a component may be deployed, each on a different node. The node at location `addr` only has access to facts f with $\pi_L(f) =$ `addr`, modeling the shared-nothing property of distributed systems.

We define a rule's **references** as the IDB relations in its body; a component references the set of relations referenced by its rules. For example, the storage component in Listing 2 references `toStorage`, `hashset`, `collisions`, and `numCollisions`. A IDB relation is an **input** of a component C if it is referenced in C and it is not in the head of any rules of C; `toStorage` is an input to the storage component. A relation that is not referenced in C but appears in the head of rules in C is an **output** of C; `fromStorage` is an output of the storage component. Note that this formulation explicitly allows a component to have multiple inputs and multiple outputs. Inputs and outputs of the component correspond to asynchronous input and output channels of each node.

Our discussion so far has been at the level of rules; we will also need to reason about individual facts. A **proof tree** [1] can be constructed for each IDB fact f, where f lies at the root of the tree, each leaf is an EDB or input fact, and each internal node is an IDB fact derived from its children via a single rule. Below we see a proof tree for one fact in `toStorage`:


Line 2
toStorage('hi', 0x7465, b.b.us:5678, 9)
Line 1
signed('hi', 0x7465, a.b.us:5678, 6)
storageNodes(b.b.us:5678)
delay(('hi', 0x7465, a.b.us:5678, 6, b.b.us:5678), 9)
in('hi', a.b.us:5678, 6)
sign('hi', 0x7465)

Listing 2. Hashset storage node in Dedalus.

```
hashset(hashed,val,l,t') :- toStorage(val,leaderSig,l,t), hash(val,hashed),
  verify(val,leaderSig), t'=t+1
hashset(hashed,val,l,t') :- hashset(hashed,val,l,t), t'=t+1
collisions(val2,hashed,l,t) :- toStorage(val1,leaderSig,l,t), hash(val1,hashed),
  hashset(hashed,val2,l,t)
numCollisions(count<val>,hashed,l,t) :- collisions(val,hashed,l,t)
fromStorage(l,sig,val,collCnt,l',t') :- toStorage(val,leaderSig,l,t),
  hash(val,hashed), numCollisions(collCnt,hashed,l,t), sign(val,sig),
  leader(l'), delay((sig,val,collCnt,l,t,l'),t')
```

2.5 Correctness

This paper transforms single-node Dedalus components into "equivalent" multi-component, multi-node Dedalus programs; the transformations can be composed to scale entire distributed protocols. For equivalence, we want a definition that satisfies any client (or observer) of the input/output channels of the original program. To this end we employ equivalence of concurrent histories as defined for linearizability [33], the gold standard in distributed systems.

We assume that a history H can be constructed from any run of a given Dedalus program P. Linearizability traditionally expects every program to include a specification that defines what histories are "legal". We make no such assumption and we consider any possible history generated by the unoptimized program P to define the specification. As such, the optimized program P' is linearizable if any run of P' generates the same output facts with the same timestamps as some run of P.

Our rewrites are safe over protocols that assume the following fault model: an asynchronous network (messages between correct nodes will eventually be delivered) where up to f nodes can suffer from general omission failures [52] (they may fail to send or receive some messages). After optimizing, one original node n may be replaced by multiple nodes $n_1, n_2, \ldots$; the failure of any of nodes n_i corresponds to a partial failure of the original node n, which is equivalent to the failure of n under general omission.

Due to a lack of space, we omit the proofs of correctness of the rewrites described in Sections 3 and 4. Full proofs, preconditions, and rewrite mechanisms can be found in the appendix of our technical report [16].

3 DECOUPLING

Decoupling partitions code; it takes a Dedalus component running on a single node, and breaks it into multiple components that can run in parallel across many nodes. Decoupling can be used to alleviate single-node bottlenecks by scaling up available resources. Decoupling can also introduce pipeline parallelism: if one rule produces facts in its head that another rule consumes in its body, decoupling those rules across two components can allow the producer and consumer to run in parallel.

Because Dedalus is a language of unordered rules, decoupling a component is syntactically easy: we simply partition the component's ruleset into multiple subsets, and assign each subset to a different node. The result is syntactically legal, but the correctness story is not quite that simple. To decouple *and* retain the original program semantics, we must address classic distributed systems

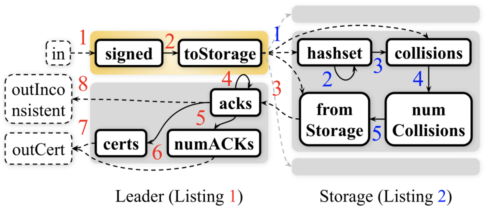


Fig. 2. Running example after mutually independent decoupling.

challenges: how to get the right data to the right nodes (space), and how to ensure that introducing asynchronous messaging between nodes does not affect correctness (time).

In this section we step through a progression of decoupling scenarios, and introduce analyses and rewrites that provably address our concerns regarding space and time. Throughout, our goal is to avoid introducing any *coordination*—i.e. extra messages beyond the data passed between rules in the original program.

General Construction for Decoupling: In all our scenarios we will consider a component C at network location addr, consisting of a set of rules $\overline{\varphi}$. We will, without loss of generality, decouple C into two components: $C_1 = \overline{\varphi}_1$, which stays at location addr, and $C_2 = \overline{\varphi}_2$ which is placed at a new location addr2. The rulesets of the two new components partition the original ruleset: $\overline{\varphi}_1 \cap \overline{\varphi}_2 = \emptyset$ and $\overline{\varphi}_1 \cup \overline{\varphi}_2 \supseteq \overline{\varphi}$. Note that we may add new rules during decoupling to achieve equivalence.

3.1 Mutually Independent Decoupling

Intuitively, if the component C_1 never communicates with C_2, then running them on two separate nodes should not change program semantics. We simply need to ensure that inputs from other components are sent to addr or addr2 appropriately.

Consider the component defined in Listing 1. There is no dataflow between the relations in Lines 1 and 2 and the relations in the remainder of the rules in the component. One possible decoupling would place Lines 1 and 2 on C_1, the remainder of Listing 1 on C_2, and reroute fromStorage messages from C_1 to C_2, as seen in Figure 2.

We now define a precondition that determines when this rewrite can be applied:

Precondition: C_1 and C_2 are mutually independent.

Recall the definition of *references* from Section 2.4: a component C references IDB relation r if some rule $\varphi \in C$ has r in its body. A component C_1 is *independent* of component C_2 if (a) the two components reference mutually exclusive sets of relations, and (b) C_1 does not reference the outputs of C_2. Note that this property is asymmetric: C_2 may still be dependent upon C_1 by referencing C_1's outputs. Hence our precondition requires *mutual* independence.

Rewrite: Redirection. Because C_2 has changed address, we need to direct facts from any relation r referenced by C_2 to addr2. We simply add a "redirection" EDB relation to the body of each rule whose head is referenced in C_2, which maps addr to addr2, and any other address to itself. For our example above, we need to ensure that fromStorage is sent to addr2. To enforce this we rewrite Line 5 of Listing 2 as follows (note variable l'' in the head, and forward in the body):

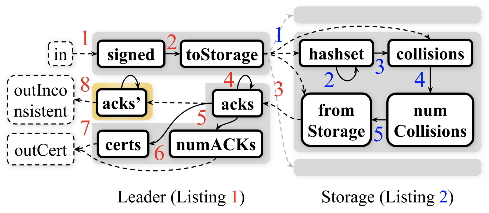


Fig. 3. Running example after monotonic decoupling.

```
fromStorage(l,sig,val,collCnt,l'',t') :- toStorage(val,leaderSig,l,t),
  hash(val,hashed), numCollisions(collCnt,hashed,l,t), sign(val,sig),
  leader(l'), forward(l',l'') delay((l,sig,val,collCnt,l,t,l''),t')
```

3.2 Monotonic Decoupling

Now consider a scenario in which C_1 and C_2 are not mutually independent. If C_2 is dependent on C_1, decoupling changes the dataflow from C_1 to C_2 to traverse asynchronous channels. After decoupling, facts that co-occurred in C may be spread across time in C_2; similarly, two facts that were ordered or timed in a particular way in C may be ordered or timed differently in C_2. Without coordination, very little can be guaranteed about the behavior of a component after the ordering or timing of facts is modified.

Fortunately, the CALM Theorem [32] tells us that *monotonic* components eventually produce the same output independent of any network delays, including changes to co-occurrence, ordering, or timing of inputs. A component C_2 is monotonic if increasing its input set from I to $I' \supseteq I$ implies that the output set $C_2(I') \supseteq C_2(I)$[1]; in other words, each referenced relation and output of C_2 will monotonically accumulate a growing set of facts as inputs are received over time, independent of the order in which they were received. The CALM Theorem ensures that if C_2 is shown to be monotonic, then we can safely decouple C_1 and C_2 without any coordination.

In our running example, the leader (Listing 1) is responsible for both creating certificates from a set of signatures (Lines 5 to 7) and checking for inconsistent ACKs (Line 8). Since ACKs are persisted, once a pair is inconsistent, they will always be inconsistent; Line 8 is monotonic. Monotonic decoupling of Line 8 allows us to offload inconsistency-checking from a single leader to the decoupled “proxy” as highlighted in yellow in Figure 3.

Precondition: C_1 is independent of C_2, and C_2 is **monotonic**.

Monotonicity of a Datalog$^{\neg}$ (hence Dedalus) component is undecidable [40], but effective conservative tests for monotonicity are well known. A simple sufficient condition for monotonicity is to ensure that (a) C_2's input relations are persisted, and (b) C_2's rules do not contain negation or aggregation. In the technical report we relax each of these checks to be more permissive.

Rewrite: Redirection With Persistence. Note that in this case we may have relations r that are outputs of C_1 and inputs to C_2. We use the same rewrite as in the previous section with one addition: we add a persistence rule to C_2 for each r that is in the output of C_1 and the input of C_2, guaranteeing that all inputs of C_2 remain persisted.

[1]There is some abuse of notation here treating C_2 as a function from one set of facts to to another, since the facts may be in different relations. A more proper definition would be based on sets of multiple relations: input and EDB relations at the input, IDB relations at the output.

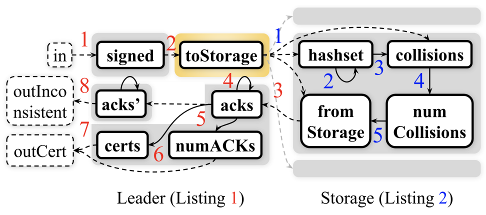


Fig. 4. Running example after functional decoupling.

The alert reader may notice performance concerns. First, C_1 may redundantly resend persistently-derived facts to C_2 each tick, even though C_2 is persistently storing them anyway via the rewrite. Second, C_2 is required to persist facts indefinitely, potentially long after they are needed. Solutions to this problem were explored in prior work [17] and can be incorporated here as well without affecting semantics.

3.3 Functional Decoupling

Consider a component that behaves like a "map" operator for a pure function F on individual facts: for each fact f it receives as input, it outputs $F(f)$. Surely these should be easy to decouple! Map operators are monotonic (their output set grows with their input set), but they are also independent per fact—each output is determined only by its corresponding input, and in particular is not affected by previous inputs. This property allows us to forgo the persistence rules we introduce for more general monotonic decoupling; we refer to this special case of monotonic decoupling as *functional decoupling*.

Consider again Lines 1 and 2 in Listing 1. Note that Line 1 works like a function on one input: each fact from `in` results in an independent signed fact in `signed`. Hence we can decouple further, placing Line 1 on one node and Line 2 on another, forwarding signed values to `toStorage`. Intuitively, this decoupling does not change program semantics because Line 2 simply sends messages, regardless of which messages have come before: it behaves like pure functions.

Precondition: C_1 is independent of C_2, and C_2 is **functional**—that is, (1) it does not contain aggregation or negation, and (2) each rule body in C_2 has at most one IDB relation.

Rewrite: Redirection. We reuse the rewrite from Section 3.1.

As a side note, recall that persisted relations in Dedalus are by definition IDB relations. Hence Precondition (2) prevents C_2 from joining current inputs (an IDB relation) with previous persisted data (another IDB relation)! In effect, persistence rules are irrelevant to the output of a functional component, rendering functional components effectively "stateless".

4 PARTITIONING

Decoupling is the distribution of *logic* across nodes; partitioning (or "sharding") is the distribution of *data*. By using a relational language like Dedalus, we can scale protocols using a variety of techniques that query optimizers use to maximize partitioning without excessive "repartitioning" (a.k.a. "shuffling") of data at runtime.

Unlike decoupling, which introduces new components, partitioning introduces additional nodes on which to run instances of each component. Therefore, each fact may be rerouted to any of the

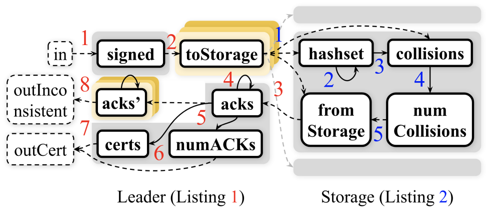


Fig. 5. Running example after partitioning with co-hashing.

many nodes, depending on the partitioning scheme. Because each rule still executes locally on each node, we must reason about changing the *location* of facts.

We first need to define partitioning schemes, and what it means for a partitioning to be correct for a set of rules. Much of this can be borrowed from recent theoretical literature [7, 27, 28, 55]. A partitioning scheme is described by a *distribution policy* $D(f)$ that outputs some node address `addr_i` for any fact f. A partitioning preserves the semantics of the rules in a component if it is **parallel disjoint correct** [55]. Intuitively, this property says that the body facts that need to be colocated remain colocated after partitioning. We adapt the parallel disjoint correctness definitions to the context of Dedalus as follows:

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1, f_2 in the proof tree of f, $D(f_1) = D(f_2)$.

Ideally we can find a single distribution policy that is parallel disjoint correct over the component in question. To do so, we need to partition each relation based on the set of attributes used for joining or grouping the relation in the component's rules. Such distribution policies are said to satisfy the co-hashing constraint (Section 4.1). Unfortunately, it is common for a single relation to be referenced in two rules with different join or grouping attributes. In some cases, dependency analysis can still find a distribution policy that will be correct (Section 4.2). If no parallel disjoint correct distribution policy can be found, we can resort to partial partitioning (Section 4.3), which replicates facts across multiple nodes.

To discuss partitioning rewrites on generic Dedalus programs, we consider without loss of generality a component C with a set of rules $\overline{\varphi}$ at network location `addr`. We will partition the data at `addr` across a set of new locations `addr1`, `addr2`, etc, each executing the same rules $\overline{\varphi}$.

4.1 Co-hashing

We begin with co-hashing [28, 55], a well studied constraint that avoids repartitioning data. Our goal is to co-locate facts that need to be combined because they (a) share a join key, (b) share a group key, or (c) share an antijoin key.

Consider two relations r_1 and r_2 that appear in the body of a rule φ, with matching variables bound to attributes A in r_1 and corresponding attributes B in r_2. Henceforth we will say that r_1 and r_2 "share keys" on attributes A and B. Co-hashing states that if r_1 and r_2 share keys on attributes A and B, then all facts from r_1 and r_2 with the same values for A and B must be routed to the same partition.

Note that even if co-hashing is satisfied for individual rules, r might need to be repartitioned *between* the rules, because a relation r might share keys with another relation on attributes A in one rule and A' in another. To avoid repartitioning, we would like the distribution policy to partition consistently with co-hashing in *every* rule of a component.

Consider Line 8 of Listing 1, assuming it has already been decoupled. Inconsistencies between ACKs are detected on a per-value basis and can be partitioned over the attribute bound to the variable `val`; this is evidenced by the fact that the relation `acks` is always joined with other IDB relations using the same attribute (bound to `val`). Line 2 and Listing 2 Line 5 are similarly partitionable by value, as seen in Figure 5.

Formally, a distribution policy D **partitions** relation r by attribute A if for any pair of facts f_1, f_2 in r, $\pi_A(f_1) = \pi_A(f_2)$ implies $D(f_1) = D(f_2)$. Facts are distributed according to their partitioning attributes.

D **partitions consistently with co-hashing** if for any pair of referenced relations r_1, r_2 in rule φ of C, r_1 and r_2 share keys on attribute lists A_1 and A_2 respectively, such that for any pair of facts $f_1 \in r_1, f_2 \in r_2$, $\pi_{A_1}(f_1) = \pi_{A_2}(f_2)$ implies $D(f_1) = D(f_2)$. Facts will be successfully joined, aggregated, or negated after partitioning because they are sent to the same locations.

Precondition: There exists a distribution policy D for relations referenced by component C that partitions consistently with co-hashing.

We can discover candidate distribution policies through a static analysis of the join and grouping attributes in every rule φ in C.

Rewrite: Redirection With Partitioning. We are given a distribution policy D from the precondition. For any rules in C' whose head is referenced in C, we modify the "redirection" relation such that messages f sent to C at `addr` are instead sent to the appropriate node of C at $D(f)$.

4.2 Dependencies

By analyzing Dedalus rules, we can identify dependencies between attributes that (1) strengthen partitioning by showing that partitioning on one attribute can imply partitioning on another, and (2) loosen the co-hashing constraint.

For example, consider a relation r that contains both an original string attribute `Str` and its uppercased value in attribute `UpStr`. The **functional dependency** (FD) `Str` $\rightarrow$ `UpStr` strengthens partitioning: partitioning on `UpStr` implies partitioning on `Str`. Formally, relation r has a *functional dependency* $g : A \rightarrow B$ on attribute lists A, B if for all facts $f \in r$, $\pi_B(f) = g(\pi_A(f))$ for some function g. That is, the values A in the domain of g determine the values in the range, B. This reasoning allows us to satisfy multiple co-hashing constraints simultaneously.

Now consider the following joins in the body of a rule: `p(str), r(str, upStr), q(upStr)`. Co-hashing would not allow partitioning, because p and q do not share keys over their attributes. However, if we know the functional dependency `Str` $\rightarrow$ `UpStr` over r, then we can partition p, q, r on the uppercase values of the strings and still avoid reshuffling. This **co-partition dependency** (CD) between the attributes of p and q loosens the co-hashing constraint beyond sharing keys. Formally, relations r_1 and r_2 have a *co-partition dependency* $g : A \hookrightarrow B$ on attribute lists A, B if for all proof trees containing facts $f_1 \in r_1, f_2 \in r_2$, we have $\pi_B(f_1) = g(\pi_A(f_2))$ for some function g. If we partition by B (the range of g) we also successfully partition by A (the domain of g).

We return to the running example to see how CDs and FDs can be combined to enable coordination-free partitioning where co-hashing forbade it. Listing 2 cannot be partitioned with co-hashing because `toStorage` does not share keys with `hashset` in Line 3. No distribution policy can satisfy

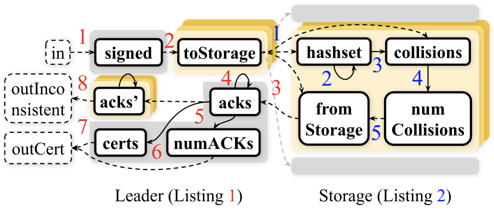


Fig. 6. Running example after partitioning with dependencies.

the co-hashing constraint if there exists two relations in the same rule that do not share keys. However, we know that *the hash is a function of the value*; there is an FD `hash.1` $\longrightarrow$ `hash.2`. Hence partitioning on `hash.2` implies partitioning on `hash.1`. The first attributes of `toStorage` and `hashset` are joined through the attributes of the `hash` relation in all rules, forming a CD. Let the first attributes of `toStorage` and `hashset`—representing a value and a hash—be V and H respectively: a fact f_v in `toStorage` can only join with a fact f_h in `hashset` if `hash`$(\pi_V(f_v))$ equals $\pi_H(f_h)$. This reasoning can be repeatedly applied to partition all relations by the attributes corresponding the repeated variable `hashed`, as seen in Figure 6.

Precondition: There exists a distribution policy D for relations r referenced in C that partitions consistently with the CDs of r.

Assume we know all CDs g over attribute sets A_1, A_2 of relations r_1, r_2. A distribution policy **partitions consistently with CDs** if for any pair of facts f_1, f_2 over referenced relations r_1, r_2 in rule φ of C, if $\pi_{A_1}(f_1) = g(\pi_{A_2}(f_2))$ for each attribute set, then $D(f_1) = D(f_2)$.

We describe the mechanism for systematically finding FDs and CDs in the technical report.

Rewrite: Identical to Redirection with Partitioning.

4.3 Partial partitioning

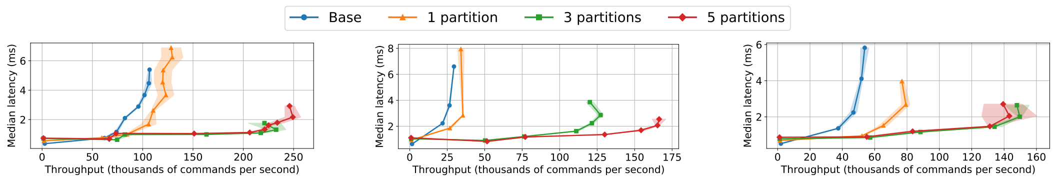
It is perhaps surprising, but sometimes additional coordination can actually help distributed protocols (like Paxos) scale.

There exist Dedalus components that cannot be partitioned even with dependency analysis. If the non-partitionable relations are rarely written to, it may be beneficial to replicate the facts in those relations across nodes so each node holds a local copy. This can support multiple local reads in parallel, at the expense of occasional writes that require coordination.

We divide the component C into C_1 and C_2, where relations referenced in C_2 can be partitioned using techniques in prior sections, but relations referenced in C_1 cannot. In order to fully partition C, facts in relations referenced in C_1 must be replicated to all nodes and kept consistent so that each node can perform local processing. To replicate those facts, inputs that modify the replicated relations are broadcasted to all nodes.

Coordination is required in order to maintain consistency between nodes with replicated facts. Each node orders replicated inputs by buffering other inputs when replicated facts f arrive, only flushing the buffer after the node is sure that all other nodes have also received f. Knowledge of whether a node has received f can be enforced through a distributed commit or consensus mechanism.

Precondition: C_1 is independent of C_2 and both behave like state machines.



(a) Voting (b) 2PC (c) Paxos

Fig. 7. Throughput/latency comparison between distributed protocols before and after rule-driven rewrites.

We define "state machines" and the rewrites for partial partitioning in the technical report.

5 EVALUATION

We will refer to our approach of manually modifying distributed protocols with the mechanisms described in this paper as *rule-driven rewrites*, and the traditional approach of modifying distributed protocols and proving the correctness of the optimized protocol as *ad hoc rewrites*.

In this section we address the following questions:

(1) How can rule-driven rewrites be applied to foundational distributed protocols, and how well do the optimized protocols scale? (Section 5.2)

(2) Which of the ad hoc rewrites can be reproduced via the application of (one or more) rules, and which cannot? (Section 5.3)

(3) What is the effect of the individual rule-driven rewrites on throughput? (Section 5.4)

5.1 Experimental setup

All protocols are implemented as Dedalus programs and compiled to Hydroflow [54], a Rust dataflow runtime for distributed systems. We deploy all protocols on GCP using n2-standard-4 machines with 4 vCPUs, 16 GB RAM, and 10 Gbps network bandwidth, with one machine per Dedalus node.

We measure throughput/latency over one minute runs, following a 30 second warmup period. Each client sends 16 byte commands in a closed loop. The ping time between machines is 0.22ms. We assume the client is outside the scope of our rewrites, and any rewrites that requires modifying the client cannot be applied.

5.2 Rewrites and scaling

We manually apply rule-driven rewrites to scale three fundamental distributed protocols—voting, 2PC, and Paxos. We will refer to our unoptimized implementations as ⓇBaseVoting, ⓇBase2PC, and ⓇBasePaxos, and the rewritten implementations as ⓇScalableVoting, ⓇScalable2PC, and ⓇScalablePaxos. In general, we will prepend the word "Base" to any unoptimized implementation, "Scalable" to any implementation created by applying rule-driven rewrites, and "Ⓡ" to any implementation in Dedalus. We measure the performance of each configuration with an increasing set of clients until throughput saturates, averaging across 3 runs, with standard deviations of throughput measurements shown in shaded regions. Since the minimum configuration of Paxos (with $f = 1$) requires 3 acceptors, we will also test voting and 2PC with 3 participants.

For decoupled-and-partitioned implementations, we measure scalability by changing the number of partitions for partitionable components, as seen in Figure 7. Decoupling contributes to the throughput differences between the unoptimized implementation and the 1-partition configuration. Partitioning contributes to the differences between the 1, 3, and 5 partition configurations.

These experimental configurations demonstrate the scalability of the rewritten protocols. They do not represent the most cost-effective configurations, nor the configurations that maximize throughput. We manually applied rewrites on the critical path, selecting rewrites with low overhead, where we suspect the protocols may be bottlenecked. Across the protocols we tested, these bottlenecks often occurred where the protocol (1) broadcasts messages, (2) collects messages, and (3) logs to disk. These bottlenecks can usually be decoupled from the original node, and because messages are often independent of one another, the decoupled nodes can then be partitioned such that each node handles a subset of messages. The process of identifying bottlenecks, applying suitable rewrites, and finding optimal configurations may eventually be automated.

Voting. Client payloads arrive at the leader, which broadcasts payloads to the participants, collects votes from the participants, and responds to the client once all participants have voted. Multiple rounds of voting can occur concurrently. ⓇBaseVoting is implemented with 4 machines, 1 leader and 3 participants, achieving a maximum throughput of 100,000 commands/s, bottlenecking at the leader.

We created ⓇScalableVoting from ⓇBaseVoting through *Mutually Independent Decoupling, Functional Decoupling*, and *Partitioning with Co-hashing*. Broadcasters broadcast votes for the leader; they are decoupled from the leader through functional decoupling. Collectors collect and count votes for the leader; they are decoupled from the leader through mutually independent decoupling. The remaining "leader" component only relays commands to broadcasters. All components except the leader are partitioned with co-hashing. The leader cannot be partitioned since that would require modifying the client to know how to reach one of many leader partitions. With 1 leader, 5 broadcasters, 5 partitions for each of the 3 participants, and 5 collectors, the maximum configuration for ⓇScalableVoting totals 26 machines, achieving a maximum throughput of 250,000 commands/s—a 2× improvement over the baseline.

2PC (with Presumed Abort). The coordinator receives client payloads and broadcasts `voteReq` to participants. Participants log and flush to disk, then reply with `votes`. The coordinator collects `votes`, logs and flushes to disk, then broadcasts `commit` to participants. Participants log and flush to disk, then reply with `acks`. The coordinator then logs and replies to the client. Multiple rounds of 2PC can occur concurrently. ⓇBase2PC is implemented with 4 machines, 1 coordinator and 3 participants, achieving a maximum throughput of 30,000 commands/s, bottlenecking at the coordinator.

We created ⓇScalable2PC from ⓇBase2PC similarly through *Mutually Independent Decoupling, Functional Decoupling*, and *Partitioning with Co-hashing*. Vote Requesters are functionally decoupled from coordinators: they broadcast `voteReq` to participants. Committers and Enders are decoupled from coordinators through mutually independent decoupling. Committers collect `votes`, log and flush commits, then broadcast `commit` to participants. Enders collect `acks`, log, and respond to the client. The remaining "coordinator" component relays commands to vote requesters. Each participant is mutually independently decoupled into Voters and Ackers. Participant Voters log, flush, then send `votes`; Participant Ackers log, flush, then send `acks`. All components (except the coordinator) can be partitioned with co-hashing. With 1 coordinator, 5 vote requesters, 5 ackers and 5 voters for each of the 3 participant, 5 committers, and 5 enders, the maximum configuration of ⓇScalable2PC totals 46 machines, achieving a maximum throughput of 160,000 commands/s—a 5× improvement.

Paxos. Paxos solves consensus while tolerating up to f failures. Paxos consists of $f + 1$ proposers and $2f + 1$ acceptors. Each proposer has a unique, dynamic ballot number; the proposer with the highest ballot number is the leader. The leader receives client payloads, assigns each payload a sequence number, and broadcasts a `p2a` message containing the payload, sequence number, and its ballot to the acceptors. Each acceptor stores the highest ballot it has received and rejects or accepts payloads into its log based on whether its local ballot is less than or equal to the leader's. The acceptor then replies to the leader via a `p2b` message that includes the acceptor's highest ballot. If this ballot is higher than the leader's ballot, the leader is preempted. Otherwise, the acceptor has accepted the payload, and when $f + 1$ acceptors accept, the payload is committed. The leader relays committed payloads to the replicas, which execute the payload command and notify the clients. 🅡BasePaxos is implemented with 8 machines—2 proposers, 3 acceptors, and 3 replicas (matching BasePaxos in Section 5.3)—tolerating $f = 1$ failures, achieving a maximum throughput of 50,000 commands/s, bottlenecking at the proposer.

We created 🅡ScalablePaxos from 🅡BasePaxos through *Mutually Independent Decoupling, (Asymmetric)*[2] *Monotonic Decoupling, Functional Decoupling, Partitioning with Co-hashing*, and *Partial Partitioning with Sealing*[3]. P2a proxy leaders are functionally decoupled from proposers and broadcast `p2a` messages. P2b proxy leaders collect `p2b` messages and broadcast committed payloads to the replicas; they are created through asymmetric monotonic decoupling, since the collection of `p2b` messages is monotonic but proposers must be notified when the messages contain a higher ballot. Both can be partitioned on sequence numbers with co-hashing. Acceptors are partially partitioned with sealing on sequence numbers, replicating the highest ballot across partitions, necessitating the creation of a coordinator for each acceptor. With 2 proposers, 3 p2a proxy leaders and 3 p2b proxy leaders for each of the 2 proposers, 1 coordinator and 3 partitions for each of the 3 acceptors, and 3 replicas, totalling 29 machines, 🅡ScalablePaxos achieves a maximum throughput of 150,000 commands/s—a 3× improvement, bottlenecking at the proposer.

Across the protocols, the additional latency overhead from decoupling is negligible.

Together, these experiments demonstrate that rule-driven rewrites can be applied to scale a variety of distributed protocols, and that performance wins can be found fairly easily via choosing the rules to apply manually. A natural next step is to develop cost models for our context, and integrate into a search algorithm in order to create an automatic optimizer for distributed systems. Standard techniques may be useful here, but we also expect new challenges in modeling dynamic load and contention. It seems likely that adaptive query optimization and learning could prove relevant here to enable autoscaling [20, 58].

5.3 Comparison to ad hoc rewrites

Our previous results show apples-to-apples comparisons between naive Dedalus implementations and Dedalus implementations optimized with rule-driven rewrites. However they do not quantify the difference between Dedalus implementations optimized with rule-driven rewrites and ad hoc optimized protocols written in a more traditional procedural language. To this effect, we compare our scalable version of Paxos to Compartmentalized Paxos [63]. We do this for two reasons: (1) Paxos is notoriously hard to scale manually, and (2) Compartmentalized Paxos is a state-of-the-art

[2]Asymmetric decoupling is defined in the technical report. It applies when we decouple C into C_1 and C_2, where C_2 is monotonic, but C_2 is independent of C_1.
[3]Partitioning with sealing is defined in the technical report. It applies when a partitioned component originally sent a batched set of messages that must be recombined across partitions after partitioning.

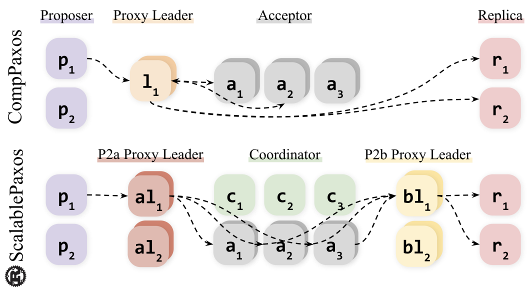

Proposer
Proxy Leader
Acceptor
Replica
CompPaxos
p_1 p_2 l_1 a_1 a_2 a_3 r_1 r_2
P2a Proxy Leader
Coordinator
P2b Proxy Leader
® ScalablePaxos
p_1 p_2 al_1 al_2 c_1 c_2 c_3 a_1 a_2 a_3 bl_1 bl_2 r_1 r_2

Fig. 8. The common path taken by CompPaxos and ®ScalablePaxos, assuming $f = 1$ and any partitionable component has 2 partitions. The acceptors outlined in red represent possible quorums for leader election.

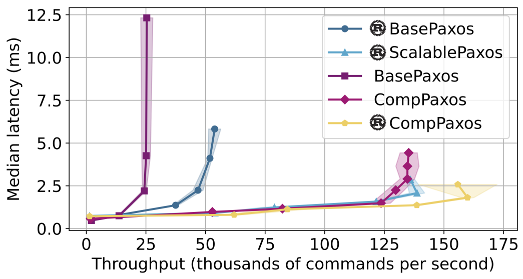


Fig. 9. Throughput/latency comparison between rule-driven and ad hoc rewrites of Paxos.

implementation of Paxos based, among other optimizations, on manually applying decoupling and partitioning.

To best understand the merits of scalability, we choose not to batch client requests, as batching often obscures the benefits of individual scalability rewrites.

5.3.1 Throughput comparison. Whittaker et al. created Scala implementations of Paxos (BasePaxos) and Compartmentalized Paxos (CompPaxos). Since our implementations are in Dedalus, we first compare throughputs of the Paxos implementations between the two languages to establish a baseline. Following the nomenclature from Section 5.2, implementations in Dedalus are prepended with ®, and implementations in Scala by Whittaker et al. are not.

BasePaxos was reported to peak of 25,000 commands/s with $f = 1$ and 3 replicas on AWS in 2021 [63]. As seen Figure 9, we verified this result in GCP using the same code and experimental setup. Our Dedalus implementation of Paxos—®BasePaxos—in contrast, peaks at a higher 50,000 commands/s with the same configuration as BasePaxos. We suspect this performance difference is due to the underlying implementations of BasePaxos in Scala and ®BasePaxos in Dedalus, compiled to Hydroflow atop Rust. Indeed, our deployment of CompPaxos peaked at 130,000 commands/s, and our reimplementation of Compartmentalized Paxos in Dedalus (®CompPaxos) peaked at a higher 160,000 commands/s, a throughput improvement comparable to the 25,000 command throughput gap between BasePaxos and ®BasePaxos.

Note that technically, CompPaxos was reported to peak at 150,000 commands/s, not 130,000. We deployed the Scala code provided by Whittaker et al. with identical hardware, network, and configuration, but could not replicate their exact result.

We now have enough context to compare the throughput between CompPaxos and ⓇScalablePaxos; their respective architectures are shown in Figure 8. CompPaxos achieves maximum throughput with 20 machines: 2 proposers, 10 proxy leaders, 4 acceptors (in a 2×2 grid), and 4 replicas. We compare CompPaxos and ⓇScalablePaxos using the same number of machines, fixing the number of proposers (for fault tolerance) and replicas (which we do not decouple or partition). Restricted to 20 machines, ⓇScalablePaxos achieves the maximum throughput with 2 proposers, 2 p2a proxy leaders, 3 coordinators, 3 acceptors, 6 p2b proxy leaders, and 4 replicas. All components are kept at minimum configuration—with only 1 partition—except for the p2b proxy leaders, which are the throughput bottleneck. ⓇScalablePaxos then scales to 130,000 commands/s, a 2.5× throughput improvement over ⓇBasePaxos. Although CompPaxos reports a 6× throughput improvement over BasePaxos from 25,000 to 150,000 commands/s in Scala, reimplemented in Dedalus, it reports a 3× throughput improvement between ⓇCompPaxos and ⓇBasePaxos, similar to the 2.5× throughput improvement between ⓇScalablePaxos and ⓇBasePaxos. Therefore we conclude that the throughput improvements of rule-driven rewrites and ad hoc rewrites are comparable when applied to Paxos.

We emphasize that our framework cannot realize every ad hoc rewrite in CompPaxos (Figure 8). We describe the differences between CompPaxos and ⓇScalablePaxos next.

5.3.2 Proxy leaders. Figure 8 shows that CompPaxos has a single component called "proxy leader" that serves the roles of two components in ⓇScalablePaxos: p2a and p2b proxy leaders. Unlike p2a and p2b proxy leaders, proxy leaders in CompPaxos can be shared across proposers. Since only 1 proposer will be the leader at any time, CompPaxos ensures that work is evenly distributed across proxy leaders. Our rewrites focus on scaling out and do not consider sharing physical resources between logical components. Moreover, there is an additional optimization in the proxy leader of CompPaxos. CompPaxos avoids relaying `p2b`s from proxy leaders to proposers by introducing `nack` messages from acceptors that are sent instead. This optimization is neither decoupling nor partitioning and hence is not included in ⓇScalablePaxos.

5.3.3 Acceptors. CompPaxos partitions acceptors without introducing coordination, allowing each partition to hold an independent ballot. In contrast, ⓇScalablePaxos can only partially partition acceptors and must introduce coordinators to synchronize ballots between partitions, because our formalism states that the partitions' ballots together must correspond to the original acceptor's ballot. Crucially, CompPaxos allows the highest ballot held at each partition to diverge while ⓇScalablePaxos does not, because this divergence can introduce non-linearizable executions that remain safe for Paxos, but are too specific to generalize. We elaborate more on this execution in the technical report.

Despite its additional overhead, ⓇScalablePaxos does not suffer from increased latency because the overhead is not on the critical path. Assuming a stable leader, p2b proxy leaders do not need to forward `p2b`s to proposers, and acceptors do not need to coordinate between partitions.

5.3.4 Additional differences. CompPaxos additionally includes classical Paxos optimizations such as batching, thriftiness [47], and flexible quorums [36], which are outside the scope of this paper as they are not instances of decoupling or partitioning. These optimizations, combined with the more efficient use of proxy leaders, explain the remaining throughput difference between ⓇCompPaxos and ⓇScalablePaxos.

5.4 On the Benefit of Individual Rewrites

In Figure 10, we examine each rewrite's scaling potential. To create a consistent throughput bottleneck, we introduce extra computation via multiple AES encryptions. When decoupling, the

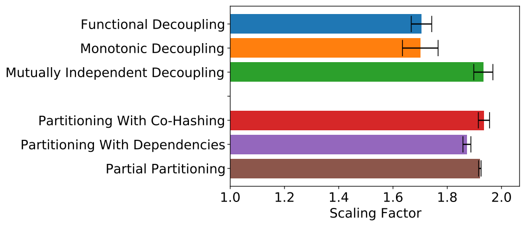


Fig. 10. The scalability gains provided by each rewrite, in isolation.

program must always decrypt the message from the client and encrypt its output. When partitioning, the program must always encrypt its output. When decoupling, we always separate one node into two. When partitioning, we always create two partitions out of one. Thus maximum scale factor of each rewrite is 2×. To determine the scaling factors, we increased the number of clients by increments of two for decoupling and three for partitioning, stopping when we reached saturation for each protocol.

Briefly, we study each of the individual rewrites using the following artificial protocols:

- *Mutually Independent Decoupling*: A replicated set where the leader decrypts a client request, broadcasts payloads to replicas, collects acknowledgements, and replies to the client (encrypting the response), similar to the voting protocol. We denote this base protocol as R-set. We decouple the broadcast and collection rules.
- *Monotonic Decoupling*: An R-set where the leader also keeps track of a ballot that is potentially updated by each client message. The leader attaches the value of the ballot at the time each client request is received to the matching response.
- *Functional Decoupling*: The same R-set protocol, but with zero replicas. The leader attaches the highest ballot it has seen so far to each response. It still decrypts client requests and encrypts replies as before.
- *Partitioning With Co-Hashing*: A R-set.
- *Partitioning With Dependencies*: A R-set where each replica records the number of hash collisions, similar to our running example.
- *Partial Partitioning*: A R-set where the leader and replicas each track an integer. The leader's integer is periodically incremented and sent to the replicas, similar to Paxos. The replicas attach their latest integers to each response.

The impact on throughput varies between rewrites due to both the overhead introduced and the underlying protocol. Note that of our 6 experiments, the first two are the only ones that add a network hop to the critical path of the protocol and rely on pipelined parallelism. The combination of networking overhead and the potential for imperfect pipelined parallelism likely explain why they achieve only about 1.7× performance improvement. In contrast, the speedups for mutually independent decoupling and the different variants of partitioning are closer to the expected 2×. Nevertheless, each rewrite improves throughput in isolation as shown in Figure 10.

6 RELATED WORK

Our results build on rich traditions in distributed protocol design and parallel query processing. The intent of this paper was not to innovate in either of those domains per se, but rather to take parallel query processing ideas and use them to discover and evaluate rewrites for distributed protocols.

6.1 Manual Protocol Optimizations

There are many clever, manually-optimized variants of distributed protocols that scale by avoiding coordination, e.g. [3, 11, 23, 39, 49, 63]. These works rely on intricate modifications to underlying protocols like consensus, with manual (and not infrequently buggy [53]) end-to-end proofs of correctness for the optimized protocol. In contrast, this paper introduces a rule-driven approach to optimization that is correct by construction, with proofs narrowly focused on small rewrites.

We view our work here as orthogonal to most ad hoc optimizations of protocols. Our rewrites are general and can be applied correctly to results of the ad hoc optimization. In future work it would be interesting to see when and how the more esoteric protocols cited above might benefit from further optimization using the techniques in this paper.

Our work was initially inspired by the manually-derived Compartmentalized Paxos [63], from which we borrowed our focus on decoupling and partitioning. Our work does not achieve all the optimizations of Compartmentalized Paxos (Section 5.3), but it achieves the most important ones, and our results are comparable in performance.

There is a long-standing research tradition of identifying commonalities between distributed protocols that provide the same abstraction [8, 10, 29, 30, 37, 60, 61, 64, 65]. In principle, optimizations that apply to one protocol can be transferred to another, but this requires careful scrutiny to determine if the protocols fit within some common framework. We attack this problem by borrowing from the field of programming languages. The language Dedalus is our "framework"; any distributed protocol expressed in Dedalus can benefit from our rewrites via a mechanical application of the rules. Although our general rewrites cannot cover every possible optimization a programmer can envision, they can be applied effectively.

6.2 Parallel Query Processing and Dataflow

A key intuition of our work is to rewrite protocols using techniques from distributed ("shared-nothing") parallel databases. The core ideas go back to systems like Gamma [22] and GRACE [25] in the 1980s, for both long-running "data warehouse" queries and transaction processing workloads [21]. Our work on partitioning (Section 4) adapts ideas from parallel SQL optimizers, notably work on auto-partitioning with functional dependencies, e.g. [70]. Traditional SQL research focuses on a single query at a time. To our knowledge the literature does not include the kind of decoupling we introduce in Section 3.

Big Data systems (e.g., [19, 38, 68]) extended the parallel query literature by adding coordination barriers and other mechanisms for mid-job fault tolerance. By contrast, our goal here is on modest amounts of data with very tight latency constraints. Moreover, fault tolerance is typically implicit in the protocols we target. As such we look for coordination-freeness wherever we can, and avoid introducing additional overheads common in Big Data systems.

There is a small body of work on parallel stream query optimization. An annotated bibliography appears in [34]. Widely-deployed systems like Apache Flink [15] and Spark Streaming [69] offer minimal insight into query optimization.

Parallel Datalog goes back to the early 1990s (e.g. [26]). A recent survey covers the state of the art in modern Datalog engines [41], including dedicated parallel Datalog systems and Datalog implementations over Big Data engines. The partitioning strategies we use in Section 4 are discussed in the survey; a deeper treatment can be found in the literature cited in Section 4 [7, 27, 28, 55].

6.3 DSLs for Distributed Systems

We chose the Dedalus temporal logic language because it was both amenable to our optimization goals and we knew we could compile it to high-performance machine code via Hydroflow. Temporal logics have also been used for *verification* of protocols—most notably Lamport's TLA+ language [44], which has been adopted in applied settings [50]. TLA+ did not suit our needs for a number of reasons. Most notably, efficient code generation is not a goal of the TLA+ toolchain. Second, an optimizer needs lightweight checks for properties (FDs, monotonicity) in the inner loop of optimization; TLA+ is ill-suited to that case. Finally, TLA+ was designed as a *finite model checker*: it provides evidence of correctness (up to k steps of execution) but no proofs. There are efforts to build symbolic checkers for TLA+ [42], but again these do not seem well-suited to our lightweight setting.

Declarative languages like Dedalus have been used extensively in networking. Loo, et al. surveyed work as of 2009 including the Datalog variants NDlog and Overlog [45]. As networking DSLs, these languages take a relaxed "soft state" view of topics like persistence and consistency. Dedalus and Bloom [5, 18] were developed with the express goal of formally addressing persistence and consistency in ways that we rely upon here. More recent languages for software-defined networks (SDNs) include NetKAT [9] and P4 [14], but these focus on centralized SDN controllers, not distributed systems.

Further afield, DAG-based dataflow programming is explored in parallel computing (e.g., [12, 13]). While that work is not directly relevant to the transformations we study here, their efforts to schedule DAGs in parallel environments may inform future work.

7 CONCLUSION

This is the first paper to present general scaling optimizations that can be safely applied to any distributed protocol, taking inspiration from traditional SQL query optimizers. This opens the door to the creation of automatic optimizers for distributed protocols.

Our work builds on the ideas of Compartmentalized Paxos [63], which "unpacks" atomic components to increase throughput. In addition to our work on generalizing decoupling and partitioning via automation, there are additional interesting follow-on questions that we have not addressed here. The first challenge follows from the separation of an atomic component into multiple smaller components: when one of the smaller components fails, others may continue responding to client requests. While this is not a concern for protocols that assume omission failures, additional checks and/or rewriting may be necessary to extend our work to weaker failure models. The second challenge is the potential liveness issues introduced by the additional latency from our rewrites and our assumption of an asynchronous network. Protocols that calibrate timeouts assuming a partially synchronous network with some maximum message delay may need their timeouts recalibrated. This can likely be addressed in practice using typical pragmatic calibration techniques.

ACKNOWLEDGEMENTS

This work was supported by gifts from AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley. http://webdam.inria.fr/Alice/pdfs/all.pdf

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367 http://arxiv.org/abs/1712.01367

[3] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 211–223. https://doi.org/10.1109/TPDS.2019.2929793

[4] Peter Alvaro, Tom J Ameloot, Joseph M Hellerstein, William Marczak, and Jan Van den Bussche. 2011. A declarative semantics for Dedalus. *UC Berkeley EECS Technical Report* 120 (2011), 2011.

[5] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings.* 249–260. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf

[6] 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*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[7] Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. 2017. Parallel-Correctness and Transferability for Conjunctive Queries. *Journal of the ACM* 64, 5 (Oct. 2017), 1–38. https://doi.org/10.1145/3106412

[8] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2022. The bedrock of bft: A unified platform for bft protocol design and implementation. *arXiv preprint arXiv:2205.04534* (2022).

[9] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

[10] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-Structured Protocols in Delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 538–552. https://doi.org/10.1145/3477132.3483544

[11] Christian Berger and Hans P Reiser. 2018. Scaling byzantine consensus: A broad analysis. In *Proceedings of the 2nd workshop on scalable and resilient infrastructures for distributed ledgers.* 13–18.

[12] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SigPlan Notices* 30, 8 (1995), 207–216.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. 2012. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 1-2 (2012), 37–51.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering* 38, 4 (2015).

[16] David C. Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites [Technical Report]. https://github.com/rithvikp/autocomp.

[17] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M Hellerstein. 2014. Edelweiss: Automatic storage reclamation for distributed programming. *Proceedings of the VLDB Endowment* 7, 6 (2014), 481–492.

[18] 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.

[19] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design I& Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 10.

[20] Amol Deshpande, Zachary Ives, Vijayshankar Raman, et al. 2007. Adaptive query processing. *Foundations and Trends® in Databases* 1, 1 (2007), 1–140.

[21] David DeWitt and Jim Gray. 1992. Parallel database systems. *Commun. ACM* 35, 6 (June 1992), 85–98. https://doi.org/10.1145/129888.129894

[22] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. 1986. GAMMA - A High Performance Dataflow Database Machine. In *VLDB.* 228–237.

[23] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[24] 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.

[25] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. 1986. An Overview of The System Software of A Parallel Relational Database Machine GRACE.. In *VLDB*, Vol. 86. 209–219.

[26] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. 1990. A framework for the parallel processing of datalog queries. *ACM SIGMOD Record* 19, 2 (1990), 143–152.

[27] Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. 2019. Parallel-Correctness and Containment for Conjunctive Queries with Union and Negation. *ACM Transactions on Computational Logic* 20, 3 (July 2019), 1–24. https://doi.org/10.1145/3329120

[28] Gaetano Geck, Frank Neven, and Thomas Schwentick. 2020. Distribution Constraints: The Chase for Distributed Data. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ICDT.2020.13

[29] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*. 363–376.

[30] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. 2023. Chemistry behind Agreement. In *Conference on Innovative Data Systems Research (CIDR).(2023)*.

[31] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 1–17.

[32] 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

[33] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[34] Martin Hirzel, Robert Soulé, Buğra Gedik, and Scott Schneider. 2018. *Stream Query Optimization*. Springer International Publishing, 1–9.

[35] Heidi Howard and Ittai Abraham. 2020. Raft does not Guarantee Liveness in the face of Network Faults. https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/.

[36] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible paxos: Quorum intersection revisited. *arXiv preprint arXiv:1608.06696* (2016).

[37] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. https://doi.org/10.1145/3380787.3393681

[38] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[39] Mohammad M Jalalzai, Costas Busch, and Golden G Richard. 2019. Proteus: A scalable BFT consensus protocol for blockchains. In *2019 IEEE international conference on Blockchain (Blockchain)*. IEEE, 308–313.

[40] Bas Ketsman and Christoph Koch. 2020. Datalog with Negation and Monotonicity. In *23rd International Conference on Database Theory (ICDT 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 155)*, Carsten Lutz and Jean Christoph Jung (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:18. https://doi.org/10.4230/LIPIcs.ICDT.2020.19

[41] Bas Ketsman, Paraschos Koutris, et al. 2022. Modern Datalog Engines. *Foundations and Trends® in Databases* 12, 1 (2022), 1–68.

[42] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[43] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[44] Leslie Lamport. 2002. Specifying systems: the TLA+ language and tools for hardware and software engineers. (2002).

[45] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009), 87–95.

[46] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

[47] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. https://doi.org/10.1145/2517349.2517350

[48] Inderpal Singh Mumick and Oded Shmueli. 1995. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence* 15 (1995), 407–435.

[49] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 35–48.

[50] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73.

[51] Diego Ongaro. 2014. *Consensus : bridging theory and practice.* Ph. D. Dissertation. Stanford University.

[52] Kenneth J. Perry and Sam Toueg. 1986. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering* SE-12, 3 (1986), 477–482. https://doi.org/10.1109/TSE.1986.6312888

[53] George Pirlea. 2023. Errors found in distributed protocols. https://github.com/dranov/protocol-bugs-list.

[54] Mingwei Samuel, Joseph M Hellerstein, and Alvin Cheung. 2021. Hydroflow: A Model and Runtime for Distributed Systems Programming. (2021).

[55] Bruhathi Sundarmurthy, Paraschos Koutris, and Jeffrey Naughton. 2021. Locality-Aware Distribution Schemes. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ICDT.2021.22

[56] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM.* ACM. https://doi.org/10.1145/3477132.3483552

[57] Pierre Sutra. 2020. On the correctness of Egalitarian Paxos. *Inform. Process. Lett.* 156 (2020), 105901. https://doi.org/10.1016/j.ipl.2019.105901

[58] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. Skinnerdb: regret-bounded query evaluation via reinforcement learning. *Proceedings of the VLDB Endowment* 11, 12 (2018), 2074–2077.

[59] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015), 36 pages. https://doi.org/10.1145/2673577

[60] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. 2015. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (July 2015), 472–484. https://doi.org/10.1109/tdsc.2014.2355848

[61] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and how to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing.* ACM. https://doi.org/10.1145/3293611.3331595

[62] Michael Whittaker. 2020. mwhittaker/craq_bug. https://github.com/mwhittaker/craq_bug.

[63] 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

[64] 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 [Technical Report]. arXiv:2012.15762 [cs.DC]

[65] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph Hellerstein, and Ion Stoica. 2021. SoK: A Generalized Multi-Leader State Machine Replication Tutorial. *Journal of Systems Research* 1, 1 (2021).

[66] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 357–368.

[67] Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. 2021. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols.. In *OSDI.* 405–421.

[68] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10).* USENIX Association, Boston, MA. https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets

[69] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles.* 423–438.

[70] Jingren Zhou, Per-Ake Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010).* IEEE, 1060–1071.

Received July 2023; accepted November 2023