Optimizing Distributed Protocols with Query Rewrites [Technical Report]

David Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky Katahanas[†], Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, Heidi Howard[‡]
UC Berkeley, Sutter Hill Ventures[†], Microsoft[‡]

ABSTRACT

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 systematically scaling *any* distributed protocol by applying rule-based rewrites, borrowing from query optimization. Distributed protocol optimizations entail a new burden: reasoning about spatiotemporal correctness. We leverage order-insensitivity and data dependency analysis enabled by Dedalus 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. Rule-based 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.

PVLDB Reference Format:
David Chu, Rithvik Panchapakesan, Shadaj Laddad,
Lucky Katahanas[†], Chris Liu, Kaushik Shivakumar, Natacha Crooks,
Joseph M. Hellerstein, Heidi Howard[‡]. Optimizing Distributed Protocols with Query Rewrites [Technical Report]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/rithvikp/autocomp.

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 [47] 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. However, each new design requires careful examination of prior work and new correctness proofs; the process is ad-hoc and error-prone [2, 36, 51, 52, 56, 61]. 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-based rewrites* that are correct by construction and can be applied to *any* distributed protocol. By enabling composable, reusable optimizations, we open the door to the creation of an automatic optimizer for distributed protocols.

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 [8], which we adopt here. It seems natural to apply query optimization in that setting. However, 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*, corresponding 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. [62] to significantly improve the throughput complex distributed protocols such as Paxos. However, while Whittaker's optimizations are handcrafted, our goal is to systematically decouple and partition *any* distributed protocol. To that end, we describe low-overhead decoupling and partitioning opportunities in distributed protocols implemented over Dedalus.

Decoupling improves scalability by spreading logic across machines to take advantage of additional physical resources and pipeline 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 classic technique from

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

query processing [23, 26]. If a partition needs to compute over non-local data, then data must be forwarded or “shuffled” across the network through asynchronous channels. Dedalus programs are composed of many rules, and it can be difficult to find a partitioning where each partition only computes over local data. We leverage relational techniques like functional dependency analysis to find data partitioning schemes that minimize communication and coordination. Again, this is a clear benefit of the relational model: functional dependencies are far easier to identify in a relational language than a procedural language.

We demonstrate the generalizability of our optimizations by systematically applying rewrites to three seminal distributed protocols: voting, 2PC, and Paxos. We specifically target Paxos [58] as it is a protocol with many distributed invariants and is challenging to verify [32, 65, 66]. The throughput of the optimized voting, 2PC, and Paxos protocols scale by 2×, 5×, and 3× respectively, a scale-up factor matching the performance of manual optimizations [62] when the underlying language of each implementation is accounted for.

Our correctness arguments focus on equivalence of localized, “peephole” optimizations of dataflow graphs. Unlike prior works, we do not need to reason about the correctness of our final optimized protocols. In fact, we do not need to reason about the specification of protocols at all; by assuming that the initial, un-optimized programs are correct, we can apply correct transformations to produce semantically equivalent programs without understanding what the program “does”.

Simultaneously, our approach to protocol optimization presents the following limitations. **(1)** Our localized (peephole) approach to optimization, which makes correctness proofs much simpler, inherently limits the space of optimizations that we consider. **(2)** Our optimizer, by design, treats the initial implementation as “law”. It cannot distinguish between true protocol invariants and implementation artifacts, thus further limiting the space of potential optimizations. **(3)** We can guarantee safety but not liveness after transformation. We show that any run of the transformed program generates the same outputs as some run of the original program, but we cannot show that all original runs are possible after transformation.

In summary, we make the following contributions:

(1) We present the preconditions and mechanisms for applying multiple correct-by-construction variants of two fundamental transformations: decoupling and partitioning.
(2) We demonstrate the application of these optimizations in complex distributed protocols such as Paxos.
(3) We evaluate our optimized programs and observe $3-5\times$ improvement in throughput across protocols, validating the role of correct-by-construction optimizations for distributed protocols.

2 BACKGROUND

Key to enabling our analysis and rewrites is our choice to implement distributed protocols in a relational logic language, Dedalus. Dedalus is a spatiotemporal logic for distributed systems [8]. 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 [25].

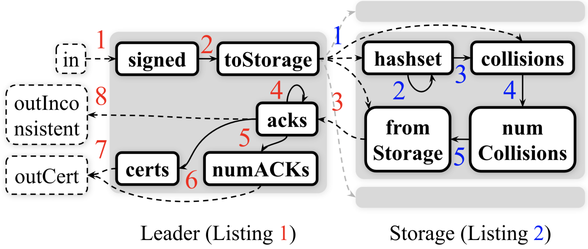


Figure 1: Dataflow diagram for a verifiably-replicated KVS. Edges are labeled with corresponding line numbers; dashed edges represent asynchronous channels. Each gray bounding box represents a node; select nodes’ dataflows are presented.

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¬ is duplicate-free.

2.1 Running example

As a running example, we focus on a verifiably replicated key-value store with hash-conflict detection inspired by [55]. 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. 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.

2.2 Datalog¬

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

A Datalog¬ **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 the rule on Line 3 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**. 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.

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 to the value of attributes *A* in fact *f*. 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¬ 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 (fixpoint) 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, negation, and fixpoints.

2.3 Dedalus

Dedalus programs are legal Datalog¬ 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" [8]) rules are captured by having the same time variable in the head literal as in the body literals. These logically "instantaneous" derivations are 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" [8]) 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` [8], `chosen` [5]), a non-deterministic function that independently maps each head fact to an arrival time. The logical formalism of the `delay` function is discussed in [5]; 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 [5]. 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 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
```

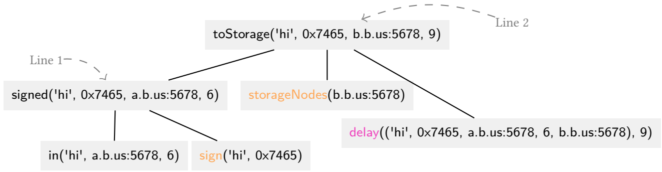
2.4 Further terminology

We introduce some additional terminology to capture the optimizations 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)

2.5 Proofs of correctness

This paper transforms single-node Dedalus components into "equivalent" multi-component, multi-node Dedalus programs; the transformations can be repeated 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 [34], the gold standard in distributed systems.

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')
```

Listing 2: Hashset storage node in Dedalus.

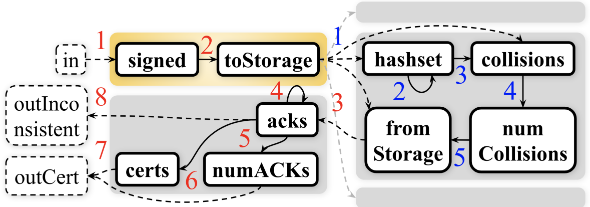
```
1  hashset(hashed,val,l,t') :-
     toStorage(val,leaderSig,l,t), hash(val,hashed),
     verify(val,leaderSig), t'=t+1
2  hashset(hashed,val,l,t') :- hashset(hashed,val,l,t),
     t'=t+1
3  collisions(val2,hashed,l,t) :-
     toStorage(val1,leaderSig,l,t), hash(val1,hashed),
     hashset(hashed,val2,l,t)
4  numCollisions(count<val>,hashed,l,t) :-
     collisions(val,hashed,l,t)
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')
```

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 as some run of P.

3 DECOUPLING

Decoupling 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



Leader (Listing 1) Storage (Listing 2)

Figure 2: Running example after mutually independent decoupling.

program semantics, we must address classic distributed systems 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 patch up 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.

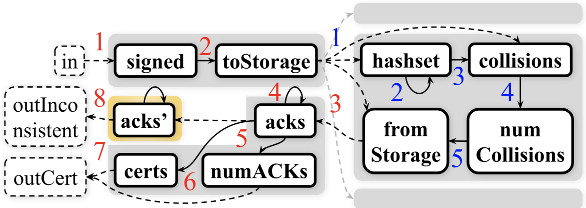
Consider the component defined in Listing 1. The relations in Lines 1 and 2 are disjoint from the relations in the remainder of the rules in the component. An effective decoupling would place Lines 1 and 2 on C_1, the remainder of Listing 1 on C_2, and reroute `fromStorage` messages from C_1 to C_2, 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



Leader (Listing 1) Storage (Listing 2)

Figure 3: Running example after monotonic decoupling.

rewrite Line 5 of Listing 2 as follows (note variable `l''` in the head, and `forward` in the body):

```
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'), forward(l',l'')
     delay((l,sig,val,collCnt,l,t,l''),t')
```

3.2 Monotonic Decoupling

Now consider a more common case, where 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.

However, the CALM Theorem [33] 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. Hence if C_2 is shown to be monotonic, then we can safely decouple C_1 and C_2 without any coordination.

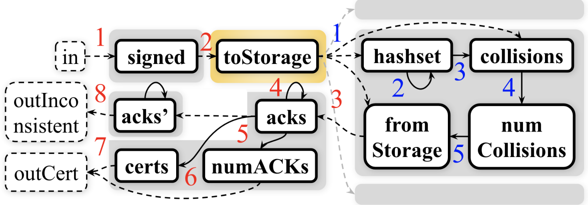
In our running example, the leader (Listing 1) is responsible for both creating certificates from a set of signatures (Lines 5 to 7) and checking for inconsistent ACKs (Line 8). Since ACKs are persisted, once a pair is inconsistent, they will always be inconsistent; Line 8 is monotonic. Monotonic decoupling of Line 8 would offload inconsistency-checking from a single leader to the decoupled "proxy" as seen 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 Appendix A.2 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

[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.



Leader (Listing 1) Storage (Listing 2)

Figure 4: Running example after functional decoupling.

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 are persisted, and C_2 remains monotonic.

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 [18] and can be incorporated here as well.

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 will allow us to forgo the persistence rules we introduce for more general monotonic decoupling; we refer to this special case 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 (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 "repartitioning" (a.k.a. "shuffling") data at runtime.

Unlike decoupling, which introduces new components, partitioning introduces additional nodes (partitions) on which to run instances of each component. Therefore, each fact may be rerouted to any of the many partitions, depending on the partitioning scheme. Because each rule still executes locally on each partition, 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 [9, 28, 29, 54]. A partitioning scheme is described by a *distribution policy* $D(f)$ that outputs some partition address `addr_i` for any fact f. A partitioning preserves the semantics of the rules in a component if it is *parallel disjoint correct* [54]. 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 resort to partial partitioning (Section 4.3), which replicates facts across multiple partitions.

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 $\overline{\varphi}$.

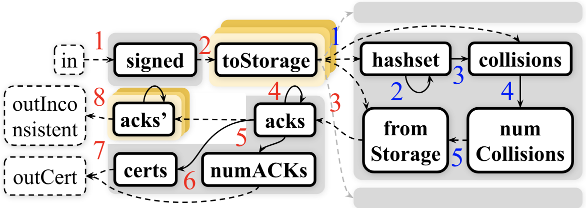
4.1 Co-hashing

We begin with co-hashing [29, 54], a well studied constraint that avoids repartitioning data. Our goal will be to co-locate facts that need to be combined because they (a) join, (b) are in the same group for aggregation, or (c) are in the same group for negation.

Co-hashing allows partitioning based on joined attributes. 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 "join" on attributes A and B. Co-hashing states that if r_1 and r_2 join 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.

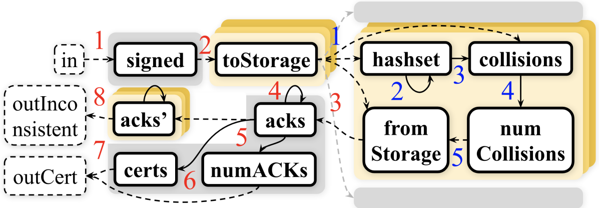
Note that even if co-hashing is satisfied for individual rules, r might need to be repartitioned *between* the rules, because a relation r might join on attributes A in one rule and A' in another. To avoid repartitioning, we would like the distribution policy to partition consistently with co-hashing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



Leader (Listing 1) Storage (Listing 2)

Figure 5: Running example after partitioning with co-hashing.



Leader (Listing 1) Storage (Listing 2)

Figure 6: Running example after partitioning with dependencies.

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, if r_1 joins r_2 on attribute lists A_1 and A_2 respectively, then 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 join 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 partition 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` ⟶ `UpStr` strengthens partitioning: partitioning on `UpStr` implies partitioning on `Str`. This reasoning allows us to satisfy multiple co-hashing constraints simultaneously.

Now consider the following joins in the body of a rule, using r from earlier: `p(str), r(str, upStr), q(upStr)`. Co-hashing would not allow partitioning, because p and q do not equi-join. However, if we know the functional dependency `Str`⟶`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 equi-joins.

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 join with `hashset` consistently in Lines 1 and 3. In Line 1, `toStorage` joins with `hashset` on its first attribute (bound to `val`), but in Line 3, `toStorage` does not join with `hashset` at all. Inconsistent joining across rules implies that there is no distribution policy satisfying the co-hashing constraint. However, we know that *the hash is a function of the value*; there is an FD `hash.1`⟶`hash.2`. Hence partitioning on `hash.2` implies partitioning on `hash.1`. The first attributes of `toStorage` and `hashset` are consistently joined through the attributes of the `hash` relation, forming a CD. This reasoning can be repeated 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.

Formally, relation r has a *functional dependency* $g: B \longrightarrow A$ on attribute lists B,A if for all facts $f \in r$, $\pi_A(f)=g(\pi_B(f))$ for some function g. That is, the values A in the domain of g determine the values in the range, B. 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 $\pi_A(f_1)=g(\pi_B(f_2))$ for some function g. Note the inversion of A and B relative to the FD definition: here we are saying if we partition by A (the range of g) we also successfully partition by B (the domain of g). 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 Appendix B.2.1.

Rewrite: Identical to Redirection with Partitioning.

4.3 Partial partitioning

It's perhaps surprising, but sometimes additional coordination can help distributed protocols (like Paxos) scale as a last resort.

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 partitions so each partition holds a local copy. This can support multiple local reads in parallel.

We divide the component C into C_1 and C_2, although unlike decoupling, C_1 and C_2 are co-located. Relations referenced in C_2 are partitioned using techniques in prior sections, but relations referenced in C_1 are not partitionable and must be replicated so each partition can perform local processing.

Coordination is required in order to maintain consistency between partitions with replicated facts. We order replicated inputs by buffering other inputs when replicated facts f arrive, only flushing the buffer after all partitions have also received f. Knowledge of whether a partition 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.

We define "state machines" in Appendix A.4 and the rewrites for partial partitioning in Appendix B.3.

5 EVALUATION

In this section we address the following questions:

(1) How can systematic optimizations be applied to foundational distributed protocols, and how well do the optimized protocols scale? (Section 5.2)

(2) Which of optimizations can be applied automatically compared to manual optimization? (Section 5.3)

(3) What is the effect of the individual systematic optimizations on throughput? (Section 5.4)

5.1 Experimental setup

All protocols are implemented as Dedalus programs and compiled to Hydroflow [53], 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 optimizations, and any optimization that requires modifying the client cannot be applied.

5.2 Optimizations and scaling

We apply systematic optimization to scale three fundamental distributed protocols—voting, 2PC, and Paxos. We will refer to our un-optimized implementations as BaseVoting, Base2PC, and BasePaxos, and the optimized implementations as ScalableVoting, Scalable2PC, and ScalablePaxos. 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 un-optimized implementation and the 1 partition configuration. Partitioning contributes to the differences between the 1, 3, and 5 partition configurations. These experiments demonstrate scalability and are not indicative of the most cost-efficient or high-throughput configurations.

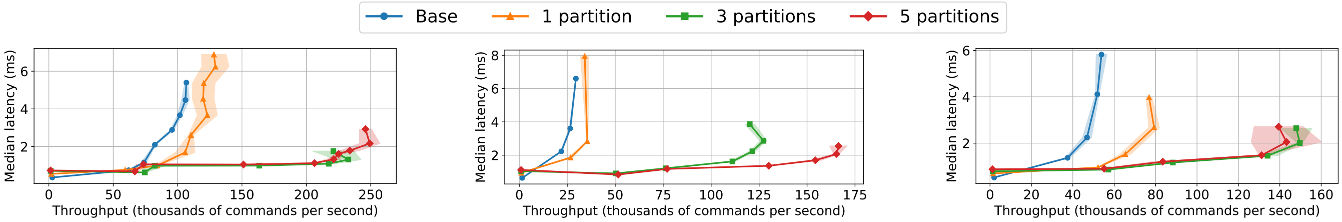
Voting. Client payloads arrive at the leader, which broadcasts payloads to the participants, collects votes from the participants, and responds to the client once all participants have voted. Multiple rounds of voting can occur concurrently. BaseVoting is implemented with 4 machines, 1 leader and 3 participants, 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 $1{+}5{+}3{*}5{+}5{=}26$ machines, ScalableVoting achieves a maximum throughput of 250,000 commands/s—a 2× improvement over the baseline, bottlenecking at the collectors.

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 flush 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 $1{+}5{+}3{*}5{+}5{+}3{*}5{+}5{=}46$ machines, Scalable2PC achieves a maximum throughput of 160,000 commands/s—a 5× improvement, bottlenecking at the committers.

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



(a) Voting

(b) 2PC

(c) Paxos

Figure 7: Throughput/latency comparison between distributed protocols before and after systematic optimization.

notify the clients. BasePaxos is implemented with 8 machines—2 proposers, 3 acceptors, and 3 replicas (matching CBasePaxos 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+2∗3+3+3∗3+2∗3+3 = 29 machines, ScalablePaxos achieves a maximum throughput of 150,000 commands/s—a 3× improvement, bottlenecking at the proposer.

Together, these experiments demonstrate that systematic optimization 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. We aim to automate this process in future work.

5.3 Comparison to manual optimization

Our previous results show apples-to-apples comparisons between naive Dedalus implementations and mechanically rewritten Dedalus implementations. However they do not quantify the difference between systematically optimized Dedalus and manually optimized code written in a more traditional procedural language. To this effect, we compare our scalable version of Paxos to Compartmentalized Paxos [62]. We do this for two reasons: (1) Paxos is notoriously hard to scale manually, and (2) Compartmentalized Paxos is a state-of-the-art 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 optimizations.

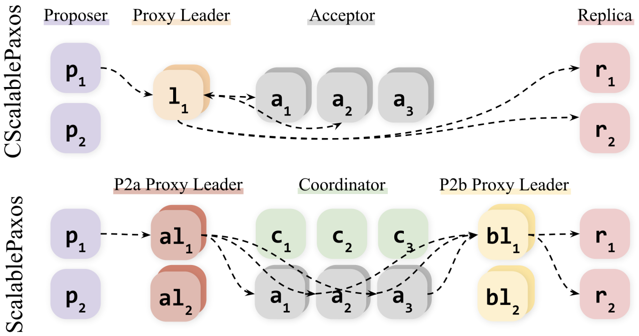


Figure 8: The common path taken by CScalablePaxos and ScalablePaxos, assuming $f = 1$ and any partitionable component has 2 partitions. The acceptors outlined in red represent possible quorums for leader election.

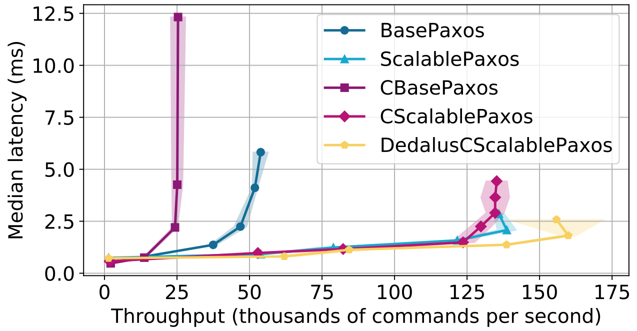


Figure 9: Throughput/latency comparison between systematic and manual optimizations of Paxos.

5.3.1 Throughput comparison. We first compare throughputs of the Paxos implementations to establish a baseline. Whittaker et al. created Scala implementations of Paxos (CBasePaxos) and Compartmentalized Paxos (CScalablePaxos). CBasePaxos was reported to peak of 25,000 commands/s with $f = 1$ and 3 replicas on AWS in 2021 [62]. 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 50,000 commands/s with the same configuration as CBasePaxos. We suspect this performance difference is due to the underlying implementations of CBasePaxos in Scala and BasePaxos in Dedalus, compiled to Hydroflow atop Rust. Indeed, our deployment of CScalablePaxos peaked at 130,000 commands/s, and our reimplementation of Compartmentalized Paxos in Dedalus (DedalusCScalablePaxos) peaked

[2] Asymmetric decoupling is defined in Appendix A.5. 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 Appendix B.4. It applies when a partitioned component originally sent sealed messages that must be recombined across partitions after partitioning.

at 160,000 commands/s, a throughput gap comparable to the 25,000 command throughput gap between CBasePaxos to BasePaxos.

Note that CScalablePaxos was reported to to peak at 150,000 commands/s. We approached but did not replicate that exact result, despite redeploying with identical hardware, network, configuration, and code.

We now have enough context to compare the throughput between ScalablePaxos and CScalablePaxos; their respective architectures are shown in Figure 8. CScalablePaxos achieves maximum throughput with 20 machines: 2 proposers, 10 proxy leaders, 4 acceptors (in a 2×2 grid), and 4 replicas. We compare CScalablePaxos 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 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 CScalablePaxos reports a 6× throughput improvement over CBasePaxos from 25,000 to 150,000 commands/s in Scala, it reports a similar 3× throughput improvement between DedalusCScalablePaxos and BasePaxos in Dedalus.

We emphasize that our framework cannot realize every optimization in CScalablePaxos (Figure 8). We describe the differences between these two protocols next.

5.3.2 *Proxy leaders.* Figure 8 shows that CScalablePaxos 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 CScalablePaxos can be shared across proposers. Since only 1 proposer will be the leader at any time, CScalablePaxos ensures that work is evenly distributed across proxy leaders. Our optimizations focus on scaling and do not colocate. Moreover, there is an additional optimization in the proxy leader of CScalablePaxos. CompPaxos avoids relaying `p2bs` 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.* CScalablePaxos 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, CScalablePaxos 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 Appendix C.

Assuming a stable leader, p2b proxy leaders do not need to forward `p2bs` to proposers, and acceptors do not need to coordinate between partitions.

5.3.4 *Additional differences.* CScalablePaxos additionally includes batching, thriftiness, and flexible quorums, 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 DedalusCScalablePaxos and ScalablePaxos.

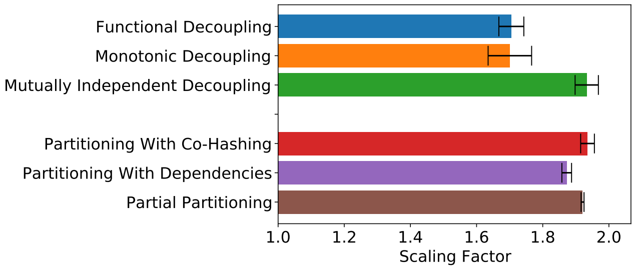


Figure 10: The scalability gains provided by each optimization, in isolation.

5.4 On the Benefit of Individual Rewrites

In Figure 10, we examine each optimization's scaling potential. To create a consistent throughput bottleneck, we introduce computation via multiple AES encryptions. When decoupling, the 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. The maximum scale factor of each optimization 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, the Dedalus programs implement the following 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 optimizations 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 optimization improves throughput in isolation as shown in Figure 10.

6 FUTURE WORK

Query optimization is classically built out of three components: transformation axioms, a cost model, and an algorithm to search the plan space. Our work here focuses on the first component in our new domain of distributed protocols.

A natural next step is to develop cost models for our context, and integrate into a search algorithm. Standard techniques may be useful here, but we also expect new challenges in modeling dynamic load and contention, and focusing on bottleneck elimination. It seems likely that adaptive query optimization and learning could prove relevant here to enable autoscaling ([21, 57], etc.)

In addition to expanding this work with more functionality, it is also natural to apply this work to new challenges, including new protocols (e.g. BFT and blockchain), and complex hardware environments. The latter is a topic of growing (manually-designed) interest in the distributed systems community [3, 45].

Finally, because our work was inspired Whittaker, et al.'s manual rewrites [62], we inherited oversights of that work: it ensures safety, but makes no strong claims about liveness or failure handling. Correctly handling those concerns in our framework would improve the robustness of our optimized protocols.

7 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 optimizations for distributed protocols.

7.1 Manual Protocol Optimizations

There are many clever, manually-optimized variants of distributed protocols that scale by avoiding coordination, e.g. [4, 13, 24, 39, 49, 62]. These works rely on intricate modifications to underlying protocols like consensus and BFS, with manual (and not infrequently buggy [52]) end-to-end proofs of correctness for the optimized protocol. In contrast, this paper introduces a systematic approach to optimization that is generalizable and correct by construction, with proofs narrowly focused on small optimization steps.

We view our work here as orthogonal to most manual optimizations of protocols. Our optimizations are general and can be applied correctly to results of the manual optimzation. 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 [62], from which we borrowed our focus on decoupling and partitioning. Our work does not automatically 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 [10, 12, 30, 31, 37, 59, 60, 63, 64]. 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 from a compiler perspective. Dedalus is our "framework"; any distributed protocol expressed in Dedalus can benefit from our optimizations automatically, without any expert oversight. Like any compiler, our work does not uncover every possible optimization a programmer can envision, but it can be quite effective in applying simple-yet-powerful optimizations, sometimes in surprisingly effective ways.

7.2 Parallel Query Processing and Dataflow

A key intuition of our work is to optimize protocols using techniques from distributed ("shared-nothing") parallel databases. The core ideas go back to systems like Gamma [23] and GRACE [26] in the 1980s, for both long-running "data warehouse" queries and transaction processing workloads [22]. Our work on partitioning (Section 4) adapts ideas from parallel SQL optimizers, notably work on auto-partitioning with functional dependencies, e.g. [69]. 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., [20, 38, 67]) 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.

There is a small body of work on parallel stream query optimization. An annotated bibliography appears in [35]; most published work is more relevant to our future agenda (Section 6) than to this paper. Widely-deployed systems like Apache Flink [17] and Spark Streaming [68] offer minimal insight into query optimization.

Parallel Datalog goes back to the early 1990s (e.g. [27]). 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 [9, 28, 29, 54].

7.3 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 [46]. As networking DSLs, these languages take a relaxed "soft state" view of topics like persistence and consistency. Dedalus and Bloom [7, 19] 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 [11] and P4 [16], but these focus on centralized SDN controllers, not distributed systems.

Further afield, DAG-based dataflow programming is explored in parallel computing (e.g., [14, 15]). 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 (Section 6).

8 CONCLUSION

This is the first paper to present systematic scaling optimizations that can be safely applied to any distributed protocol. We open the door to the creation of automatic optimizers for distributed protocols.

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] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond consensus for microsecond applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation.* 599–616.

[4] 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

[5] 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.

[6] 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

[7] 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

[8] 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.

[9] 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

[10] 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).

[11] 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.

[12] 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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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).

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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

[25] 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.

[26] 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.

[27] 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.

[28] 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

[29] 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

[30] 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.

[31] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. [n.d.]. Chemistry behind Agreement. ([n. d.]).

[32] 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.

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

[34] 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.

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

[36] 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/.

[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))*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. 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] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering.. In *OSDI*. 467–483.

[46] 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.

[47] 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.

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

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

[54] 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

[55] 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

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

[57] 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.

[58] 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

[59] 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

[60] 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

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

[62] 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

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

[64] 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).

[65] 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.

[66] 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.

[67] 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

[68] 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.

[69] 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.

A DECOUPLING

We will require the following terms in addition to the terms introduced in Section 2.

An **instance** I over program P is a set of facts for relations in P. An **immediate consequence** operator evaluates rules to produce new facts from known facts. $T_\varphi(I)$ over instance I and rule φ is a set of facts f_h :- $f_1,...,f_n$ that is an instantiation of φ, where each f_i is in I. For the remainder of this paper, when we refer to instance, we mean an instance created by evaluating a sequence of immediate consequences over some set of EDB and input facts. An instance is the state of the Dedalus program as a result of repeated rule evaluation.

A relation r' is in the proof tree of r if there exists facts $f' \in r'$ and $f \in r$ such that f' is in the proof tree of f.

We assume that there is no *entanglement* [8]; for any fact, values representing location and time only appear in the location and time attributes respectively. This allows us to modify the location and times of facts without worrying about changing the values in other attributes. Our transformations may introduce entanglement when necessary.

A.1 Mutually independent decoupling

A.1.1 Proof. Let I be an instance over C and I' be an instance over C_1 and C_2.

We will demonstrate that for each I', there exists an I such that (1) for relations r referenced in C_1 and output relations of C_1 and C_2, I contains fact f in r if and only if I' contains f, and (2) for relations r referenced in C_2, I contains f in r if and only if I' contains f', where f and f' share the same values except $\pi_L(f') =$ `addr2` and $\pi_L(f') =$ `addr`. By showing that I' implies the existence of I with the same input and output facts, each history H' constructed from I' must then be equivalent to some history H constructed by I, completing our proof.

We prove by induction; we assume that our proof is correct for any instance I' constructed through n immediate consequences, and show that it holds for $n+1$. The base case of 0 immediate consequences is trivial; I and I' start with the same set of EDB facts.

Let $T_\varphi(I')$ be the n+1-th immediate consequence over I'.

Consider φ where φ is a rule of C_1. Relations r in $body(\varphi)$ are by definition referenced in C_1, so by the induction hypothesis, the same facts exist in r for both I and I'. Therefore the same immediate consequence is possible over both I and I', producing the same fact at the head of φ, proving the inductive case.

Now consider φ where φ is a rule of C_2. For each r in the body of φ, I and I' share the same facts except those facts f in I have $\pi_L(f) =$ `addr` and f' in I' have $\pi_L(f') =$ `addr2`. The same immediate consequence is possible in both I and I' differing only in the location of facts, assuming the location attribute is only used for joins (as enforced by Dedalus). If φ is synchronous, then the fact at the head of φ retains its location value in both I and I', proving the inductive case. If φ is asynchronous, then the head of φ must be an output relation with the same location value in both I and I'.

Relations r with facts generated in C_1 and used in C_2 must be forwarded from `addr` to `addr2` through an asynchronous message channel. In Dedalus, this can be achieved by modifying synchronous rules into asynchronous rules, turning r into an output relation of C_1 and an input relation of C_2.

A.2 Monotonic decoupling

A.2.1 Checks for monotonicity. The strictest check for monotonicity requires that (1) all input relations of C are persisted, and that (2) no rules of C contain aggregation or negation constructs. Fundamentally, these preconditions imply that the existence of any fact f guarantees the fact f' will exist at $\pi_T(f') = \pi_T(f) + 1$ and f' otherwise equals f. If all facts f over a relation r has this property, then we say r is **logically persisted**. Any persisted relation (with a persistence rule) must be logically persisted, but a logically persisted relation is not necessarily persisted. We relax the preconditions for logical persistence below.

A relation r is logically persisted if all the relations r is dependent on are logically persisted. Consider the proof tree of any fact f in r, and all children facts f_c in r_c. If all r_c are logically persisted, then f_c' must exist where $\pi_T(f_c') = \pi_T(f_c) + 1$ and f_c' otherwise equals f_c. Since the rules of C are monotonic (no aggregation or negation), then those facts alone are enough to guarantee the existence of f' where $\pi_T(f') = \pi_T(f) + 1$ and f' otherwise equals f. Therefore r must be logically persisted.

Threshold operations over monotone lattices are also monotonic [19], despite involving aggregations.

To constrain the memory footprint of monotonic components over time, we allow facts to be garbage collected from persisted relations through user annotations. For example, in Paxos, whether a quorum is reached is a monotonic condition. Once a quorum is reached for a particular sequence number in Paxos, the committed value cannot change, so the votes for that quorum can be safely forgotten (no longer persisted). To allow monotonic components to forget values, we allow the user to annotate persistence rules with garbage collecting conditions.

A.2.2 Mechanism. Monotonic decoupling employs both the Redirection rewrite (Section 3.1) and the Decoupling rewrite (Appendix A.3.1) in addition to the following rewrite to persist inputs to C_2:

Monotonic Rewrite: For all input relations r' of C_2:

- Create a relation r'' with all the attributes of r' and replace all references of r' in C_2 with r''.
- Add the alias and persistence rules to C_2:

```
r''(...,l,t) :- r'(...,l,t)
r''(...,l,t') :- r''(...,l,t), t'=t+1
```

A.2.3 Proof. The proof is similar to that of Appendix A.3.2 for φ where φ is a rule of C_1. We defer to the CALM Theorem [33] for φ where φ is a rule of C_2.

A.3 Functional decoupling

A.3.1 Mechanism. Functional decoupling employs the Redirection rewrite (Section 3.1) to route data from outside C' to C_2. Routing data from C_1 to C_2 requires the explicit introduction of asynchrony below.

Rewrite: Decoupling. Given a rule φ in C_1 with a head relation r referenced in C_2:

- Create a relation r' with all the attributes of r, and replace all references of r in C_2 with r'.
- Add the forwarding rule to C_1:

```
r'(...,l',t') :-
   r(...,l,t), forward(v,l'), delay((...,l,t,l'),t')
```

A.3.2 Proof. Formally, we will demonstrate that for each instance I' over C_1 and C_2, there exists an instance I over C such that (1) for relations r referenced in C_1 and output relations of C_1 and C_2 (excluding input relations of C_2), I contains fact f in r if and only if I' contains f, (2) for relations r referenced in C_2, I contains fact f in r if and only if I' contains f', where f and f' share the same values except $\pi_L(f) =$ `addr` while $\pi_L(f') =$ `addr2`, and $\pi_T(f) \leq \pi_T(f')$.

We again prove by induction, assuming the proof is correct for I' through n immediate consequences, and show that it holds for n+1. The base case is trivial.

Let $T_\varphi(I')$ be the n+1-th immediate consequence over I'.

Consider φ' where φ' is a rule of C_1. If φ' is unchanged from φ in C, then the inductive hypothesis implies the same facts in both I and I' in all relations in the body of φ', so $T_\varphi(I')$ implies $T_\varphi(I)$.

If φ' is is a newly asynchronous rule, then let r be the head of φ' and f' be the fact of r in $T_{\varphi'}(I')$. Let φ be the original, synchronous rule. r must be an input relation of C_2, so we must show that f' is at the head of $T_{\varphi'}(I')$ if and only if f is at the head of $T_\varphi(I)$, where $\pi_L(f) =$ `addr`, $\pi_L(f') =$ `addr2`, and $\pi_T(f') > \pi_T(f)$. Let t be the time and l be the location of all body facts in $T_{\varphi'}(I')$. We know t and l are also the time and location of all body facts in $T_\varphi(I)$ by the inductive hypothesis. φ' differs from φ with the two additional relations `forward` and `delay` added to the body. `forward` assigns f' the location `addr2`, while `delay` sets the time of f' to some nondeterministic value greater than t. Since the original rule φ was synchronous, f shares the same location `addr` as all other facts in $T_\varphi(I)$, and either time t (if φ is deductive) or t+1 (if φ is inductive). This proves the inductive hypothesis: f and f' share the same values except $\pi_L(f) =$ `addr` while $\pi_L(f') =$ `addr2`, and $\pi_T(f) \leq \pi_T(f')$.

Now consider φ' where φ' is a rule of C_2. φ' in C_2 is unchanged from φ in C. Since we assumed that C_2 is functional, φ' contains at most one IDB relation in its body. If there are only EDB relations in its body, then the facts in those EDBs are the same in both I' and I, completing the proof. If there is one IDB relation r_b in its body, then by the induction hypothesis, the fact f_b' of r_b in I' implies fact f_b in I, where $\pi_L(f_b) =$ `addr` and $\pi_L(f_b') =$ `addr2`, $\pi_T(f_b) < \pi_T(f_b')$, and f_b' otherwise equals f_b. All remaining relations in the body of φ must be EDBs with the same facts across all locations and times. Therefore, any immediate consequence $T_{\varphi'}(I')$ with f_b' in its body and f' in its head implies $T_\varphi(I)$ with f_b in its body and f in its head. If φ is synchronous, then f' and f_b' share the same location and time, f and f_b share the same location and time, and f' and f are otherwise equal, completing the proof.

If φ is asynchronous, then f' and f are output facts. Then f' and f share the same location (the destination), whereas for time, the facts only need to satisfy the inequalities $\pi_T(f') > \pi_T(f_b')$ and $\pi_T(f) > \pi_T(f_b)$. In other words, the range of possible values for $\pi_T(f')$ is $(\pi_T(f_b'),\infty)$ and the range of possible values for $\pi_T(f)$ is $(\pi_T(f_b),\infty)$. Since $\pi_T(f_b') > \pi_T(f_b)$, the range $(\pi_T(f_b'),\infty)$ must be a sub-range of $(\pi_T(f_b),\infty)$. Therefore, given f' in $T_{\varphi'}(I')$, an immediate consequence $T_\varphi(I)$ with f is always possible where $\pi_T(f) = \pi_T(f')$ and $f = f'$, completing the proof.

A.4 State machine decoupling

Although this decoupling technique has since been cut from the paper, we still include it since its preconditions and proofs are referenced in Appendix B.3.

In a state machine component, any pair of facts that are combined (say via join or aggregation) at time t must be the result of inputs at time t and the order of inputs prior, but the exact value of t is irrelevant. In these cases, we want to guarantee that (a) facts that co-occur at time t in C will also be processed together at some time t' in C_2, and (b) the inputs of C prior to t match the inputs of C_2 prior to t'. To meet this guarantee, we collect facts from C_1 to C_2 into sequenced batches.

Precondition: C_1 is independent of C_2, and C_2 behaves like a **state machine**.

Before we formalize what it means to behave like a state machine, a couple of definitions are helpful.

Definition A.1 (Existence dependency)**.** Relation r has an *existence dependency* on input relations $\overline{r_{in}}$ if r is empty whenever there is no input; that is, $r = \emptyset$ in any timestep when $\bigwedge_{r_i \in \overline{r_{in}}} r_i = \emptyset$.

Definition A.2 (No-change dependency)**.** Relation r has a *no-change dependency* on input relations $\overline{r_{in}}$ if r's contents remain unchanged in a timestep when the inputs are empty. That is, if $\bigwedge_{r_i \in \overline{r_{in}}} r_i = \emptyset$ at timestep t, then r contains exactly the same facts at timestep t as it did in timestep $t-1$.

Formally, C_2 is a state machine if (a) all referenced relations have either existence or no-change dependencies on the inputs, and (b) outputs of C_2 have existence dependencies on the inputs. Condition (a) ensures that the component is insensitive to the passing of time(steps), and (b) ensures that the passing of time(steps) does not affect output content.

A.4.1 Checks for state machine behavior.

We provide conservative tests on relations to identify existence and no-change dependencies. A relation r has an existence dependency on input relations $\overline{r_{in}}$ if for all rules φ in the proof tree of r, (1) φ does not contain `t'=t+1`, and (2) the body of φ contains at least one non-negated relation r' where either r' is an input or r' also has an existence dependency on $\overline{r_{in}}$.

A relation r has a no-change dependency on input relations $\overline{r_{in}}$ if:

(1) **Explicit persist.** If there is an inductive rule φ with $r = head(\varphi)$, φ must be the persistence rule. Then r is persisted.
(2) **Implicit persist.** If is no such inductive rule, then for all (non-inductive) rules φ where $r = head(\varphi)$, the body of φ contains only EDBs and relations r' where r' has a no-change dependency on $\overline{r_{in}}$.
(3) **Change only on inputs.** If there is such an inductive rule, then we also allow rules φ where $r = head(\varphi)$ to contain at least one non-negated relation r' in the body, where either $r' \in \overline{r_{in}}$ or r' has an existence dependency on $\overline{r_{in}}$.

A.4.2 Mechanism.

To guarantee coexistence of facts, rewrites for state machine decoupling must preserve the order and batching of inputs. Similar to the rewrites above, we create new relations and asynchronous forwarding rules for relations in C_1 referenced in C_2. To preserve ordering and batching of inputs, we create additional rules in C_1 to track the number of previous batches and the current batch size (the number of output facts to C_2). Then C_2 ensures that all previous batches have been processed and the current batch has arrived before processing any input fact in the current batch.

Unlike any decoupling techniques described so far, we cannot reroute rules from other components C' to C_2; those input facts must be batched and ordered by C_1. Intuitively, if all inputs are routed through C_1, then the batching and ordering on C_1 is a feasible batching and ordering on C. C_2 must then process its inputs with the same batching and ordering to guarantee correctness. Were facts f to arrive at C_2 without batching or ordering information from C_1, then f may happen-after some input f' in C_1 but be processed before f' is processed at C_2, violating causality.

Our rewrites append a new attribute T_1 to relations forwarded to C_2 from C_1. This attribute represents the time on C_1 when each input fact existed, allowing C_2 to process facts in the same order and batches as C_1 even with non-deterministic message delay.

Rewrite: Batching. Given a rule φ in either C_1 or another component C' whose head r is referenced in C_2, we add the following rules to C_1:

```
# Create a relation r′ with all the attributes of r,
   with an additional attribute T1, and forward to C2.
r'(...,t,l',t') :- r(...,l,t), forward(l,l'),
   delay((...,l,t,l'),t')
# Count the number of facts of r for any time.
   Assumes that count evaluates to 0 if r is empty.
rCount(count<...>,l,t) :- r(...,l,t)
# Sum the size of the batch across relations ri.
batchSize(n,l,t) :- r1Count(n1,l,t), r2Count(n2,l,t),
   ..., n=n1+n2+...
# Record whenever the batch size is non-zero.
batchTimes(t,l,t') :- batchSize(n,l,t), n!=0, t'=t+1
batchTimes(t,l,t') :- !batchSize(n,l,t),
   batchTimes(t,l,t), t'=t+1
# Send the batch size and times to C2.
inputs(n,t,prevT,l',t') :- batchTimes(prevT,l,t),
   batchSize(n,l,t), n!=0, forward(l,l'),
   delay((n,t,prevT,l,l'),t')
inputs(n,t,0,l',t')
   :- !batchTimes(prevT,l,t), batchSize(n,l,t),
   n!=0, forward(l,l'), delay((n,t,prevT,l,l'),t')
```

On C_2, we modify all references of input relations r to `rSealed` and add the following rules:

```
r''(...,t1,l,t') :- r'(...,t1,l,t)
# Count the number of facts of r for batch t1.
r''(...,t1,l,t') :- r''(...,t1,l,t), t'=t+1
rCount(count<...>,t1,l,t) :- r''(...,t1,l,t)
# Count the number of facts across r_i for batch t1.
recvSize(n,t1,l,t) :- r1Count(n1,t1,l,t),
   r2Count(n2,t1,l,t), ..., n=n1+n2+...
# Check if this batch has been
   received and the previous batch has been processed.
inputs(n,t1,prevT,l,t') :- inputs(n,t1,prevT,l,t),
   t'=t+1
canSeal(t1,l,t) :- recvSize(n,t1,l,t),
   inputs(n,t1,prevT,l,t), sealed(prevT,l,t)
canSeal(t1,l,t)
   :- recvSize(n,t1,l,t), inputs(n,t1,0,l,t)
# Mark this batch as processed.
sealed(t1,l,t') :- canSeal(t1,l,t), t'=t+1
sealed(t1,l,t') :- sealed(t1,l,t), t'=t+1
# Can process facts at time t1.
rSealed(...,l,t) :- r''(...,t1,l,t), canSeal(t1,l,t)
```

Note that whenever a time $t1$ is sealed on C_2, facts in r'' and inputs can be garbage collected. Facts in sealed can be garbage collected if a higher timestamp has been sealed. We omit these optimizations for simplicity.

A.4.3 Proof. Our proof relies on C_2 processing inputs in the same order and batches as C_1. For simplicity, we denote $I_{\overline{r},t}$ as the set of facts f in I where f is a fact of relation r in $\overline{r}$ and $\pi_T(f)=t$.

Formally, we will prove that for each instance I' there exists I such that (1) for relations r referenced in C_1 and output relations of C_1 and C_2 (excluding input relations of C_2), I contains fact f in r if and only if I' contains f, and (2) for the set of relations $\overline{r'}$ referenced in C_2 (and $\overline{r}$ corresponding to relations with rSealed replaced with r), for any time t' where at least one input relation rSealed of C_2 is not empty in I' and contains fact f'_{in}, let $t=\pi_{T_1}(f'_{in})$. We must have $I'_{\overline{r'},t'}=I_{\overline{r},t}$ when facts in rSealed are mapped to r, and location and time are ignored.

For rules φ' of C_1, the inductive proof is identical to previous proofs.

Now consider φ' where φ' is a rule of C_2 and φ' corresponds to φ in C. Let t' be the time of immediate consequence $T_{\varphi'}(I')$ such that for all facts f' of relations r in the body of φ', $\pi_T(f')=t'$. If all input relations (rSealed, not r) are empty for t' in I', then I' cannot produce output facts at t', since output relations must have existence dependencies on the input relations.

If at least one input relation rSealed is not empty for t' in I', we must show $I'_{\overline{r'},t'}=I_{\overline{r},t}$. Let $t=\pi_{T_1}(f'_{in})$ for some fact f'_{in} in rSealed with $\pi_T(f'_{in})=t'$. Let $t'_<$ be the time of the previous input on I'; formally, for all input relations rSealed of C_2, there is no t'' where $t'_< < t'' < t'$ such that $I'_{\texttt{rSealed},t''}$ is non-empty. Similar to how we construct t from t', let $t_< = \pi_{T_1}(f'_{in})$ for some fact f'_{in} in rSealed with $\pi_T(f'_{in})=t'_<$.

We first show that $t_<$ is the time of the previous input on I; formally, for input relations rSealed of C_2, for all corresponding r, there is no t'' where $t_< < t'' < t$ such that $I_{r,t''}$ is non-empty. We prove by contradiction, assuming such t'' exists. In order for a fact f'_{in} in rSealed in I' to have $\pi_T(f'_{in})=t'$, canSeal must contain the fact canSeal(t, addr2, t'), which is only possible if sealed contains the fact sealed(t'',addr2,t') and inputs contains the fact inputs(n,t,t'',addr2,t'). sealed(t'',addr2,t') implies the fact canSeal(t'',addr2,$t'_<$) where $t'_< = \pi_T(f'_{in})$. In order for some fact f in r to join with canSeal(t'',addr2,$t'_<$) to create f_{in}, we must have $\pi_{T_1}(f)=t''=\pi_{T_1}(f_{in})$. By definition, $\pi_{T_1}(f_{in})=t'_<$, so $t'_<=t''$.

Knowing that no inputs facts exist with times between $t_<$ and t in I and between $t'_<$ and t' in I', we can use the existence and no-change dependencies of relations r in I to reason about the instances $I'_{\overline{r'},t'}$ and $I_{\overline{r},t}$. By the induction hypothesis, we have $I'_{\overline{r'},t'_<} = I_{\overline{r},t_<}$. We can now reason about the instances I' and I at times $t'-1$ and $t-1$, respectively. For all relations r with existence dependencies on the inputs, $I'_{r,t'-1}$ and $I_{r,t-1}$ must both be empty. For all relations r with no-change dependencies on the inputs, $I'_{r,t'-1}=I'_{r,t'_<}$ and $I_{r,t-1}=I_{r,t_<}$, so $I'_{r,t'-1}=I_{r,t-1}$.

Consider an immediate consequence $T_{\varphi'}(I')$ with facts f' in the body of φ' where $\pi_T(f')=t'$. We find the set of facts in the proof tree of f' that have no proof tree (because they are inputs or EDBs) or have parents in the tree with time $t'-1$. Inputs and EDBs are the same between I' and I at time t' and t, due to our sealing mechanism. Since we know $I'_{r,t'-1}=I_{r,t-1}$, any parent fact in the tree with time $t'-1$ exists in I with time $t-1$ and evaluates to the same fact. Therefore, the same series of immediate consequences are possible in I to produce each fact in $I'_{r,t'-1}$. Thus $I'_{\overline{r'},t'}=I_{\overline{r},t}$.

If φ' is an asynchronous rule, then the head of φ' is an output relation r, and we have to show that the fact f' in r' of immediate consequence $T_{\varphi'}(I')$ is equivalent to some fact f in r of immediate consequence $T_{\varphi}(I)$. Output relations must have existence dependencies in C_2 by assumption, so given f_b in the body of $T_{\varphi'}(I')$ with $\pi_T(f)=t'$, the input relations are not empty at t' in I', and $I'_{\overline{r'},t'}=I_{\overline{r},t}$. f'_b of φ' for I' must correspond to f_b in the body of φ for I, $\pi_L(f_b)=$ addr and $\pi_T(f_b)=t$ instead of $\pi_L(f'_b)=$ addr2 and $\pi_T(f'_b)=t'$. The facts f' and f at the head of φ' and φ must be the same as well, non-deterministic with the constraints $\pi_T(f') > t'$ and $\pi_T(f) > t$. We know $t' > t$ due to the addition of an asynchronous channel, therefore $\pi_T(f)=\pi_T(f')$ is always possible, and $f=f'$, completing the proof.

A.5 Asymmetric decoupling

In this section, we consider decoupling where C_1 and C_2 are mutually dependent and where C_2 is independent of C_1 instead.

If C_1 and C_2 are both monotonic, then they can be decoupled through the Redirection With Persistence rewrite (Section 3.2), according to the CALM Theorem [33].

Now consider C_2 independent of C_1, where C_2 exhibits useful properties for decoupling. Intuitively, although C_2 forwards facts to C_1, we can treat the time in which C_1 processes inputs as the "time of input arrival" while allowing C_2 to process inputs first.

This presents a problem: C_2 might produce outputs "too early", violating well-formedness. To preserve well-formedness, we introduce a rewrite to delay output facts f' derived from input fact f of C_2 until C_1 acknowledges it has processed f.

Precondition: C_2 is independent of C_1, and C_2 is (1) a state machine and (2) either monotonic or functional.

A.5.1 Mechanism. To decouple, first apply the Batching rewrite (Appendix A.4.2) for all rules in C_2 whose head r is referenced in C_1, replacing r with `rSealed`. Populate `forward` with both `forward(addr,addr2)` and `forward(addr2,addr)`. Then apply the forwarding rewrite for all rules in another component C' whose head is referenced in C_2. Finally, perform the following rewrite to delay outputs of C_2:

Rewrite: Batch Acknowledgement. Add the following rules to track which batches C_1 has processed:

```
# Component C1.
batchACK(t2,l',t')
   :- canSeal(t2,l,t), forward(l,l'), delay(t,t')
# Component C2.
batchACK(t2,l,t') :- batchACK(t2,l,t), t'=t+1
```

For each rule φ in C_2 with output relation `out`, create the relation `outP` with two additional attributes— $T2$ and L'—representing the derivation time of the fact and the destination, and add the following rules:

```
# Replace φ with this rule.
outP(t,l',...,l,t) :- ...
# Find the batchACK each output must wait on.
outBatchTime(max<prevT>,l,t)
   :- batchTimes(prevT,l,t),
   outP(t2,l',...,l,t), prevT <= t2
# Outputs derived at time t2 can now send.
outCanSend(t2,l,t)
   :- outBatchTime(t2,l,t), batchACK(t2,l,t)
# Buffer outputs until they can send.
outP(t2,l',...,l,t')
   :- outP(t2,l',...,l,t), t'=t+1, !outCanSend(t2,l,t)
# Send the outputs.
out(...,l',t') :-
   outP(t2,l',...,l,t), outCanSend(t2,l,t), delay(t,t')
```

A.5.2 Proof. Since input relations r to C_1 are now buffered and replaced with `rSealed`, the arrival time of input facts in the transformed component C no longer corresponds to the processing time. This poses a problem for our proof; in the original component C, the arrival time of input facts *is* the processing time. Note that since messages are sent over an asynchronous network, any entity that sends and receives messages from C can only observe the "send" time t_s of each input fact f, where $t_s < \pi_T(f)$. Intuitively, an input fact that is sent at time t_s and in-network for t seconds is processed identically to a fact sent at time t_s, in-network for t' seconds, and buffered for b seconds, so long as $t'+b=t$.

To formalize this intuition, we introduce the **observable instance** $\mathcal{I}$, which contains a set of facts representing the send times of input facts and arrival times of output facts. An instance I is **observably equivalent** to $\mathcal{I}$ if for all facts f in relation r of I, (1) if r is an input relation, then f exists if and only if there exists f_s in $\mathcal{I}$ where f equals f_s except $\pi_T(f) > \pi_T(f_s)$, and (2) if r is an output relation, then there exists f in I if and only if there exists f in $\mathcal{I}$.

The history of an instance I is constructed using the wall-clock times of inputs and outputs of its observably equivalent instance $\mathcal{I}$. Therefore, any instances I_1 and I_2 that are observably equivalent to the same $\mathcal{I}$ share the same histories.

We will prove that for each instance I' there exists I and the observable instance $\mathcal{I}$ such that (1) I and I' are both observably equivalent to $\mathcal{I}$, and (2) Each fact f of relation r referenced by C_1 in I' exists if and only if f exists in I.

We select the time of input facts in I such that its inputs are observably equivalent to $\mathcal{I}$. For each input relation r in C: (1) If r is an input of C_1, each fact f of r in I' exists if and only if f exists in I, and (2) If r is an input of C_2, given f' of r in I' with $t'=\pi_T(f')$, let the seal time be $t_s=\pi_{T2}(f_s)$ in f_s of `canSeal` in I'; let f equal f' except $\pi_L(f)=$ `addr` and $\pi_T(f)=\pi_T(f_s)$; f' exists if and only if f exists in I, and (3) If no such f_s exists, then let $\pi_T(f)=\pi_T(f')$. Note that in case 2, $\pi_T(f) > \pi_T(f')$ due to asynchrony from C_2 to C_1. Therefore, input facts f in I correspond to input facts f' in I' where $\pi_T(f) \geq \pi_T(f')$, thus the inputs of I are observably equivalent to $\mathcal{I}$.

We first prove the claim that I' and I are equivalent for all facts over relations referenced in C_1. We prove by induction over the immediate consequence of rules φ whose head r is referenced in C_1. If φ is an original rule of C_1, then by the inductive hypothesis, the claim is true. If φ is a rule of another component C', then the rule is unchanged and the claim is still true.

Now consider φ' where the head of φ' is `rSealed` corresponding to some r after transformation. φ' is a rule we introduced so there is no immediate consequence over it in I. Let φ refer to the rule with r at its head instead of `rSealed`; φ is a rule of C_2. We can show that the immediate consequence $T_\varphi(I')$ at time t' is possible if and only if $T_\varphi(I)$ at some time $t>t'$ is possible; this is true if for all relations $r_b \in body(\varphi)$, $I'_{r_b,t'} = I_{r_b,t}$. Then as long as $t > t'$, the facts of r of immediate consequence $T_\varphi(I')$ can be asynchronously delivered from C_2 to C_1 at some time t_d, where $t \geq t_d > t'$, and be sealed at time t in a an immediate consequence $T_{\varphi'}(I')$ over φ', resulting in the same facts in `rSealed` in I' and r in I.

Since C_2 behaves like a state machine, the facts of r_b are dependent on the input facts of C_2 and their ordering. Since C_2 is either functional or monotonic, the facts of r_b are not dependent on the ordering of the input facts; this can be proven by treating each r_b as an output relation, then reapplying proofs from Appendices A.2.3 and A.3.2. Let $\overline{r}$ be the input relations of C_2. We can prove that facts f' of $\overline{r_{in}}$ in I' exists if and only if f of $\overline{r_{in}}$ in I exists, where f' equals f except $\pi_L(f')=$ `addr2` while $\pi_L(f)=$ `addr`, and if $\pi_T(f')=t'$ then $\pi_T(f)=t$, but if $\pi_T(f')<t'$, then $\pi_T(f)<t$. In other words, I and I' share inputs at t' and t and all previous inputs, but previous inputs may be out-of-order.

We prove by contradiction. First consider some f_i' of an input relation of C_2 in I' where $\pi_T(f_i')=t'$, but there is no f_i in I where $\pi_T(f_i)=t$. Since r an output of C_2 and input of C_1, it is batched according to the batching rewrite, and there must be a fact f_c in `canSeal` of I' with $\pi_{T2}(f_c)=t'$ signalling when facts in r can be processed. By construction of I, f_i' exists in I' if and only if f_i exists in I, a contradiction. Now consider some f_i' where $\pi_T(f_i')<t'$, but there is no f_i where $\pi_T(f_i)<t$. Either there is some f_c as above, or $\pi_T(f_i)=\pi_T(f_i')$ by construction of I. Since $t'<t$, $t'=\pi_T(f_i)<t$, completing the proof. Therefore, for inputs and outputs of C_1, I' and I and observably equivalent to $\mathcal{I}$.

We now prove that I' and I are observably equivalent to $\mathcal{I}$ for inputs and outputs of C_2. By construction of I, the inputs of I are observably equivalent to $\mathcal{I}$. Since C_2 is either functional or monotonic, and C_2 is independent of C_1, given the same input facts of C_2 in I' and I, in any relation r referenced in C_2, f' exists in r of I' if and only if f exists in I, where f' equals f except on time. Consider output fact f' derived at time t' in I' and t in I. In I', f' is buffered in `outP` until the latest input fact of C_2 is acknowledged by C_1. Since we assumed that C_2 is a state machine, all outputs of C_2 must have existence dependencies on its inputs, and the latest input fact f_i' of C_2 in I' must have time t'. There must be a fact f_c in `canSeal` of I' with $\pi_{T2}(f_c) = t'$. By construction of I, $t = \pi_T(f_c)$. The acknowledgement for f_i must be sent at t in I' and arrive at some time $t_o > t$ at C_2, when f' can be sent. Therefore, the range of possible times of f is (t,∞) and (t_o,∞) for f', where the range of f' is a sub-range of f, and any immediate consequence of f' in I' must be possible in I.

B PARTITIONING

B.1 Partitioning by co-hashing

B.1.1 Mechanism. After finding a distribution policy D that partitions consistently with co-hashing, the partitioning rewrite routes input facts to partitions `addr_i` by injecting the distribution policy D. We model D as a relation in Dedalus, such that if $D(f) =$ `addr_i`, then we add the tuple `D(..., addr_i)`, where `...` represents the values of f, excluding time: We then apply the following rewrite:

Rewrite: Redirection With Partitioning. Given a rule φ in another component C' whose head relation r is referenced in C:

- Add the body term $D(\ldots, l')$ to φ, where ... is bound to the variables in the head of the rule.
- Replace the location variable v of the head of the rule with l'. Replace v with l' in `delay` similarly.

Note that this rewrite differs from the Redirection rewrite (Section 3.1) in that the entire fact is used to determine the new destination, while the Redirection rewrite only considers the original destination.

B.1.2 Proof. Formally, we will prove that for each instance I' over the component C partitioned by co-hashing with distribution policy D, there exists I over the original component C such that (1) for output relations r, I contains fact f in r if and only if I' contains f, and (2) for input relations r or relations referenced in C, I contains f in r if and only if I' contains f', where f and f' share the same values except $\pi_L(f') = D(f)$.

First consider φ' where the head relation r of φ' is an input of C, and φ' is a rule of some other component C'. Let φ be the original rule. Facts in the body of φ and φ' are the same in both I' and I, and the partitioning rewrite only changes the location of f' in the head of φ' such that $\pi_L(f) = D(f)$. Thus our claim holds for input relations of C.

Now consider φ in C. All relations in the body of φ are referenced by C, by definition.

Consider synchronous or inductive φ, such that the head relation r of φ is referenced in C as well; we need to prove that fact f of r in $T_\varphi(I)$ exists if and only if f' exists in $T_{\varphi'}(I')$, where $\pi_L(f') = D(f')$. Assume by induction that this holds for I and I'. φ equals φ' since the rewrite does not alter rules in C. For relation $r_h = head(\varphi)$ and any $r_b \in body(\varphi)$, there must be some attributes A_h,A_b that r_h,r_b join on; otherwise D cannot exist and we do not partition. Let f_b be the fact of r_b in $T_\varphi(I)$, and f_b' be the corresponding fact in $T_\varphi(I')$. D must partition consistently with co-hashing on A_h,A_b, so $D(f_h) = D(f_b)$ and $D(f_h') = D(f_b')$. φ is synchronous or inductive, so we must have $\pi_L(f_h') = \pi_L(f_b')$. The inductive hypothesis states that $D(f_b') = \pi_L(f_b')$. Since $D(f_h') = D(f_b')$ and $\pi_L(f_h') = \pi_L(f_b')$, we must have $D(f_h') = \pi_L(f_h')$. Therefore the induction hypothesis holds for all instances I and I'.

If φ is asynchronous, relations $r \in body(\varphi)$ must be inputs or referenced relations. By the proofs above, facts f_1,f_2 of any pair of relations $r_1, r_2 \in body(\varphi)$ are partitioned consistently with co-hashing on the attributes they join on. Therefore $D(f_1) = D(f_2)$, implying $\pi_L(f_1') = \pi_L(f_2')$ for the corresponding f_1',f_2' which are otherwise identical to f_1,f_2. Therefore any immediate consequence over these facts f_1,f_2 of $T_\varphi(I)$ can always correspond to some immediate consequence $T_\varphi(I')$ over f_1',f_2', and vice versa. The location of the head (the output relation) is unmodified, completing our proof.

Note that we do not separately prove correctness for aggregation and negation because they are covered by our proofs according to the definition of "join" in Section 4.1.

B.2 Partitioning with dependencies

B.2.1 Checks for dependencies. FDs in Dedalus can be created in three ways:

- **EDB annotation.** For example, `hash(M, H)` is the EDB relation that simulates the hash function $hash(m) = h$, so there is an FD from the attributes M to H where $g(x) = hash(x)$.
- **Variable sharing.** For a relation r, if in all rules φ with r as the head, attributes A,B of r always share a variable, then there is an FD from A to B and from B to A, where $g(x) = x$.
- **Inheritance.** The heads of rules φ can inherit functional dependencies from a combination of joined relations in the body of φ.

In the last case, to determine which dependencies are inherited, we perform the following analysis for each relation r in the head of rule φ:

- **Attribute-variable substitution.** Take the set of all FDs of all relations in the body of φ and replace each domain/co-domain on attributes with their bound variables in φ.
- **Constant substitution.** If an attribute is bound to a constant instead of another variable, plug the constant into the FDs of that attribute. Now all FDs in φ should be functions on variables.
- **Transitive closure.** Construct the transitive closure of all such FDs.
- **Variable-attribute substitution.** Replace each FD on variables with their bound attributes from r, if possible. The FDs that only contain attributes in r are the possible FDs of r.

Having described the process of extracting FDs for each relation r at the head of each rule φ, we must determine which FDs hold *across* rules. Since the identification of FDs for any relation assumes that all dependent relations have already been analyzed, and Dedalus allows dependency cycles, FD analysis must be recursive. We divide the process of identifying FDs for r into two steps: union and intersection. The union step recursively takes the union of generated dependencies in any rule φ where r is the head; the intersection step recursively removes FDs that are not generated in some rule φ where r is the head.

CDs can be similarly extracted using dependency analysis. In the variable-attribute substitution step, instead of only retaining FDs where variables in the domain and co-domain can *all* be replaced with attributes in r, retain FDs where *any* variable can be replaced with attributes in r. These are the CDs of r and the relations r joins with in φ; they describe how attributes of r joins with other relations in φ. The CDs that hold across rules can be identified with the intersection step for FDs.

B.2.2 Mechanism. The rewrite mechanics are identical to those in Appendix B.1.1.

B.2.3 Proof. The proofs are similar to those in Appendix B.1.2, assuming a CD g exists over attributes A,B of relations r_1,r_2 only if for any f_1, f_2 of r_1, r_2 in the same proof tree, we must have $\pi_A(f_1) = g(\pi_B(f_2))$.

B.3 Partial partitioning

B.3.1 Mechanism. In order to replicate relations r referenced in C_1, we introduce a relation `partitions`(p) and populate `partitions` with a tuple for each partition location $addr_i$.

Rewrite: Replication. Given a rule φ in another component C' whose head relation r is referenced in C:

- If r is referenced in C_1, add the body term `partitions`(p) to φ and bind the location attribute of the head to p.
- Otherwise, apply the partitioning rewrite.

Add the following rules to C:

```
processedI(i,l,t') :- processedI(i,l,t), t'=t+1
maxProcessedI(max<i>,l,t) :- processedI(i,l,t)
maxReceivedI(max<i>,l,t) :- receivedI(i,l,t)
unfreeze(l,t) :- maxReceivedI(i,l,t),
   maxProcessedI(i,l,t), !outstandingVote(l,t)
```

As we show below, `unfreeze(l,t)` will be appended to rules so they can only be executed when all previous replicated inputs have been processed.

For each relation r referenced in C_1, replace r with `rSealed` and add the following rules:

```
r(...,l,t') :- r(...,l,t), t'=t+1
# Send replicated messages to the proxy for ordering.
rVote(l,...,l',t')
   :- r(...,l,t), proxy(l,l'), delay((...,l,t,l'),t')
# The proxy
   sends rCommit when all partitions have sent rVote.
rCommit(i,...,l,t') :- rCommit(i,...,l,t), t'=t+1
receivedI(i,l,t) :- rCommit(i,...,l,t)
# Messages in
   rCommit are processed in the proxy-assigned order.
rSealed(next,...,l,t) :-
   maxProcessed(i,l,t), next=i+1, rCommit(next,...,l,t)
rSealed(0,...,l,t)
   :- !maxProcessed(i,l,t), rCommit(0,...,l,t)
processedI(i,l,t') :- rSealed(i,...,l,t), t'=t+1
outstandingVote(l,t)
   :- r(...,l,t), !rCommit(i,...,l,t)
```

For each remaining relation r in C, replace r with `rSealed`, and add the following rules:

```
r(...,l,t') :- r(...,l,t), t'=t+1, !unfreeze(l,t)
rSealed(...,l,t) :- r(...,l,t), unfreeze(l,t)
```

We describe the functionality of the proxy node but omit its implementation. The proxy node receives messages from each partition through `rVote` for each relation r and provides assigns each fact in `rVote` a unique, incrementing order. Once the same message has been sent to the proxy from all partitions, the proxy broadcasts the ordered message through `rCommit`.

B.3.2 Proof. Before stating our proof goal, we present terms to describe partitioned state. For simplicity, we denote $I_{\overline{r},p,t}$ as the set of facts f in I where f is a fact of a relation in $\overline{r}$, $D(f) = p$, and $\pi_T(f) = t$. A relation r is *empty* at time t in instance I and partition p if there is no fact f of r in I where $\pi_T(f) = t$ and $D(f) = p$. The *corresponding facts* of a replicated fact f are all facts f' where f equals f' except $\pi_L(f) \neq \pi_L(f')$.

Since input relations r to C_1 are now buffered and replaced with `rSealed`, the arrival time of input facts in the transformed component C no longer corresponds to the processing time. This poses a problem for our proof; in the original component C, the arrival time of input facts *is* the processing time. We relax this requirement through the observation that because messages are sent over an asynchronous network, any entity that sends and receives messages from C can only observe the "send" time t_s of each input fact f, where $t_s < \pi_T(f)$. Intuitively, an input fact that is sent at time t_s and in-network for t seconds is processed identically to a fact sent at time t_s, in-network for t' seconds, and buffered for b seconds, so long as $t' + b = t$.

We will prove that for each instance I' over the partially partitioned component C, there exists I over the original component C and the observable instance $\mathcal{I}$ such that (1) I and I' are both observably equivalent to $\mathcal{I}$, and (2) for the set of relations $\overline{r}$ referenced in C (and $\overline{r'}$ corresponding to $\overline{r}$ with `rSealed` replacing r), for any time t' and partition p where at least one input relation `rSealed` of C_1 is not empty in I' and contains fact f', let f be the fact in I corresponding to f' with the smallest time $t = \pi_T(f)$, and let $p = \pi_L(f')$. We must have $I'_{\overline{r'},p,t'} = I_{\overline{r},p,t}$. In other words, although each replicated input is delivered at different times at different partitions, the states of the partitions at the times of delivery correspond to the original state at a single time of input delivery. (3) Similarly, if at least one input relation `rSealed` of C_2 is not empty, then the same condition holds with $t' = t$, $p = D(f')$.

First, observe that claim 1 holds if (a) each output fact f of C exists in I' if and only if f is also in I, and (b) claims 2 and 3 hold. Let $\mathcal{I}$ be observably equivalent to I; each input fact f of r in I exists if and only if there exists f_s in $\mathcal{I}$ where f equals f_s except $\pi_T(f) > \pi_T(f_s)$. By claims 2 and 3, there must be f' in I' where f' equals f except $\pi_T(f') \geq \pi_T(f)$ (since $\pi_T(f)$ is based on the smallest $\pi_T(f')$ across partitions), which implies $\pi_T(f') > \pi_T(f_s)$. If output facts in I and I' are identical, then I' must also be observably equivalent to $\mathcal{I}$.

Claims 2 and 3 imply that output facts in I' are also possible in I. Output relations in C are assumed to have existence dependencies on inputs, so given the inputs f' and f above, if the range of possible times for any output fact f'_o in I' is $(\pi_T(f'),\infty)$, then the range of possible times for the same output facts f_o in I must be $(\pi_T(f),\infty)$. Since $\pi_T(f') \geq \pi_T(f)$, so the range of possible output times of f'_o must be a sub-range of f_o, and any f'_o in I' is possible in I.

Therefore it suffices to prove claims 2 and 3.

We select the time of input facts in I such that its inputs are observably equivalent to $\mathcal{I}$. For facts f' in I' over relations `rSealed` in C: (1) If `rSealed` is referenced in C_1, let f'_a be the corresponding fact of f' in I' with the smallest time $t' = \pi_T(f')$, and let f equal f'_a except $\pi_L(f) =$ `addr`; f of the corresponding r exists in I if and only if f' exists in I'. (2) If `rSealed` is referenced in C_2, let f equal f' except $\pi_L(f') =$ `addr`; f of the corresponding r exists in I if and only if f' exists in I'. By the partial partitioning rewrite, each fact in `rSealed` contains a fact f_r in r in its proof tree with an earlier time, which must be later than the send time of f_r. Therefore, since the times of input facts f in I is set to the times of facts in `rSealed`, $\pi_T(f)$ must be later than the send time of f, and the inputs of I are observably equivalent to $\mathcal{I}$.

We now prove claim 2 and 3 by induction on facts with time t_i in I'.

We show that claim 2 holds for facts with time $t' = t_i + 1$, assuming claims 2 and 3 hold up to t_i. We will prove by induction of the inputs in I' and I up to time t'. Let f' be a fact in input relation `rSealed` of C_1 in I' with $t' = \pi_T(f')$, $p = \pi_L(f')$, and f'_a be the fact in I' corresponding to f' with the smallest time $t = \pi_T(f'_a)$. Let f equal f'_a except $\pi_L(f) =$ `addr`. By definition of I above, f is an input fact in I. By the partial partitioning rewrite, there is no other co-occurring input fact f'' in I' with $\pi_T(f'') = t'$, since `processedI` will not contain the index of `rCommit` until the next timestep. Therefore, I does not contain any other input fact at t. Let f'_1 be the most recent fact in any input relation r_1 in I' with $t'_1 = \pi_T(f'_1)$, $t' > t'_1$, and $p = \pi_L(f'_1)$, such that there does not exist f'_2 input fact with $t'_2 = \pi_T(f'_2)$ in I' where $t' > t'_2 > t'_1$. Let f'_{1a} correspond to f'_1 with the smallest time $t_1 = \pi_T(f'_{1a})$. Let f_1 equal f'_{1a} except $\pi_L(f_1) =$ `addr`. By definition of I above, f_1 is an input fact in I. By the inductive hypothesis, we have $I'_{\overline{r'},p,t'_1} = I_{\overline{r},p,t_1}$.

Since no input facts exist in I' for partition p between t'_1 and t', if we can show that no input facts exist in I for partition p between t_1 and t, then we can reuse the proof from Appendix A.4.3 combined with the partitioning proof from Appendix B.1.2 to show that $I'_{\overline{r'},p,t'} = I_{\overline{r},p,t}$.

Lemma 1 (Order consistency). *Given facts f_1, f_2 of an input relation of C_1 in I, $\pi_T(f_1)$ is less than $\pi_T(f_2)$ if and only if for each partition p, for each pair of corresponding facts f'_1, f'_2 in I' where $\pi_L(f'_1) = \pi_L(f'_2) = p$, we have $\pi_T(f'_1) < \pi_T(f'_2)$.*

Proof. Assume by contradiction that for some partition p, $\pi_T(f'_1) \geq \pi_T(f'_2)$. By the partial partitioning rewrite, we know that $\pi_T(f'_1) = \pi_T(f'_2)$ is impossible for input relations of C_1. Let the order $O(f')$ of a fact f' in an input relation `rSealed` of C_1 in I' be the value of the index attribute of its parent fact in `rCommit`. Then $\pi_T(f'_1) > \pi_T(f'_2)$ if and only if $O(f'_1) > O(f'_2)$, which holds across all partitions p by the partial partitioning rewrite. Now let f'_{1a} be the corresponding fact of f_1 with the smallest time in I', and f'_{2b} for f_2. By construction of I, $\pi_T(f'_{1a}) = \pi_T(f_1)$ and $\pi_T(f'_{2b}) = \pi_T(f_2)$. Let f'_{2a} be the corresponding fact of f_2 where $\pi_L(f'_{2a}) = \pi_L(f'_{1a})$; f'_{1a} and f'_{2a} are the corresponding inputs for f_1 and f_2 on one specific partition. Define f'_{1b} for f'_{2b} similarly. Since ordering is consistent across partitions, $O(f'_1) > O(f'_2)$ implies $O(f'_{1a}) > O(f'_{2a})$, which implies $\pi_T(f'_{1a}) > \pi_T(f'_{2a})$. Since $\pi_T(f_1) < \pi_T(f_2)$, we know $\pi_T(f'_{1a}) < \pi_T(f'_{2b})$, therefore $\pi_T(f'_{2a}) < \pi_T(f'_{2b})$. This contradicts the definition of f'_{2b} as the fact in I' corresponding to f_2 with the smallest time, since f'_{2a} corresponds to f_2 with a smaller time. Proof by contradiction. □

Assume for contradiction that there exists f_2 of r_2 in I, $t_2 = \pi_T(f_2)$, $t > t_2 > t_1$, such that the most recent fact of I' does not correspond to the most recent fact of I. Let f'_2 be the corresponding input fact in I' with $t'_2 = \pi_T(f'_2)$, $p = \pi_L(f'_2)$ such that $t' > t'_2$.

If r_2 and r_1 are both inputs of C_1, then by Lemma 1, $t > t_2 > t_1$ implies $t' > t'_2 > t'_1$, which contradicts the definition of f'_1 as the most recent input fact.

If r_2 is an input of C_1 and r_1 is an input of C_2, then $t > t_2$ implies $t' > t'_2$, and $t_1 = t'_1$. Since $t_2 > t_1$ and $t'_2 \geq t_2$ by construction of I, we have $t' > t'_2 > t'_1$ which is again a contradiction.

Now consider r_2 as an input of C_2 such that $t'_2 = t_2$. If r_1 is an input of C_1, then let f'_{1a} be the corresponding input fact in I' with the smallest time $t'_{1a} = \pi_T(f'_{1a})$. By definition, $t'_1 \geq t'_{1a}$. By construction of I, $t'_{1a} = t_1$, therefore $t > t_2 > t_1$ implies $t' > t'_2 > t'_{1a}$. By the partial partitioning rewrite, no input facts can arrive on partition p between t'_{1a} and t'_1, since the `outstandingVote` relation will force input relations to buffer. Therefore, given $t'_2 > t'_{1a}$ and $t'_1 \geq t'_{1a}$, we must have $t'_2 > t'_1 \geq t'_{1a}$. Combined with $t' > t'_2 > t'_{1a}$, we now have $t' > t'_2 > t'_1 \geq t'_{1a}$; $t'_2 > t'_1$ contradicts the definition of f'_1 as the most recent input fact.

Otherwise, if r_1 is an input of C_2, then $t'_1 = t_1$. Then $t_2 > t_1$ implies $t'_2 > t'_1$, which contradicts the definition of f'_1 as the most recent input fact.

We've proven that the most recent input fact f'_1 in I' up to t' corresponds to the most recent input fact f_1 in I up to t, and we can reuse earlier proofs to prove claim 2.

We show claim 3 holds, using the same variable definitions. By the partial partitioning rewrite, we know that if an input relation `rSealed` of C_2 is not empty, then input relations `rSealed` of C_1 must be empty. Let f' be the input fact in I' at time t' with corresponding $\underline{f}$ in I at t. Note that this proof can be generalized to a set of facts $\overline{f'}$ at time t'. By construction of I, $t' = t$. Again assuming the most recent input fact f'_1 of r_1 in I' at time t'_1 corresponds to f_1 in I at t_1, we show that there is no f_2 of r_2 in I at t_2 where $t > t_2 > t_1$.

If r_2 and r_1 are both inputs of C_1, then $t_2 > t_1$ implies $t'_2 > t'_1$ by Lemma 1. Since $t = t'$, so $t > t_2 > t_1$ implies $t' > t'_2 > t'_1$, which contradicts the definition of f'_1 as the most recent input fact.

If r_2 is an input of C_1 and r_1 is an input of C_2, then $t_1 = t'_1$. Since $t'_2 \geq t_2$ by construction of I, $t_2 > t_1$ implies $t'_2 > t'_1$, and since $t = t'$, $t > t_2 > t_1$ implies $t' > t'_2 > t'_1$, a contradiction.

If r_2 is an input of C_2 and r_1 is an input of C_1, then $t_2 = t'_2$. Since $t = t'$ and $t > t_2$, $t' > t'_2$. $t' > t'_2 > t'_1$ is a contradiction, and $t'_2 \neq t'_1$ by the partial partitioning rewrite, so we must have $t' > t'_1 > t'_2$. Let f'_{1a} correspond to f_1 in I' with the smallest time $t'_{1a} = \pi_T(f'_{1a})$. By construction of I, $t'_{1a} = t_1$. By definition, $t'_1 \geq t'_{1a}$. By the partial partitioning rewrite, no input facts can arrive on partition p between t'_{1a} and t'_1. Therefore, given $t'_1 \geq t'_{1a}$ and $t'_1 > t'_2$, we must have $t'_1 \geq t'_{1a} > t'_2$. $t'_{1a} > t'_2$ implies $t_1 > t_2$, which contradicts our assumption that $t > t_2 > t_1$.

If r_2 and r_1 are both inputs of C_2, then $t_2 = t'_2$ and $t_1 = t'_1$. Combined with $t = t'$, $t > t_2 > t_1$ implies $t' > t'_2 > t'_1$ which is a contradiction.

By induction on previous facts of input relations, we also have $I'_{\overline{r'},p,t'_1} = I_{\overline{r},p,t_1}$. Since inputs are the same between I' and I at time t' and t, we can similarly use the proofs from Appendices A.4.3 and B.1.2 to prove claim 3, completing the proof of correctness.

B.4 Partitioning sealing

B.4.1 Sealing. Sealing [6] is a syntactic sugar we introduce to simulate sending multiple output facts in a single asynchronous message. Sealed relations can be partially partitioned so each partition can compute and send its own fraction of the sealed outputs.

Syntactically, seal is added to the head of a rule φ using aggregation syntax. We demonstrate how to seal the relation r in the output relation out of C below, where s and u are additional illustrative relations:

```
1   # Send rule on component C.
2   out(seal<r>,a,l',t') :- r(...,l,t),
      s(a,l,t), dest(l',l,t), delay((...,a,l,t,l'),t')
3   # Receive rule on component C′.
4   u(...,a,l,t) :- out(...,a,l,t)
```

This desugars into the following, where out is replaced with outSealed in C':

```
1   # Component C.
2   rCount(count<...>,a,l,t) :- r(...,l,t), s(a,l,t)
3   outCount(c,a,l',t') :- rCount(c,a,l,t), s(a,l,t),
      dest(l',l,t), delay((c,a,l,t,l'),t')
4   out(...,a,l',t') :- r(...,l,t), s(a,l,t),
      dest(l',l,t), delay((...,a,l,t,l'),t')
5   # Component C′.
6   outReceived(count<...>,a,l,t) :- out(...,a,l,t)
7   sealed(a,l,t) :- outReceived(c,a,l,t),
      outCount(c,a,l,t)
8   u(...,a,l,t) :- out(...,a,l,t), sealed(a,l,t)
9   # Only persist until sealed.
10  out(...,a,l,t')
      :- out(...,a,l,t), !sealed(a,l,t), t'=t+1
11  outCount(c,a,l,t')
      :- outCount(c,a,l,t), !sealed(a,l,t), t'=t+1
```

The sugared syntax for sealing guarantees that the relations rCount, outCount, outReceived, etc are only used as described above, and correctness is preserved as long as after partial partitioning, the facts in outSealed match the facts originally in out.

B.4.2 Mechanism. Sealing can be partitioned through dependency analysis on the sugared syntax; among the rules φ introduced in C, all the joins in the body of φ were already present in the sugared syntax.

If out cannot be partitioned with dependencies, and it has an existence dependency on some input relation in of C_1, then the rewrite is as follows:

Rewrite: Partitioning Sealing. After completing the partial partitioning rewrite over the sugared syntax, perform the following rewrites. On C, replace Lines 1 and 4 with the following, assuming inCommit is defined for in as specified in Appendix B.3.1:

```
1   outCount(l,i,c,a,l,t) :- rCount(c,a,l,t),
      inCommit(i,...,l,t), dest(l',l,t),
      delay((l,i,c,a,l,t,l'),t')
2   out(i,...,a,l',t')
      :- r(...,a,l,t), s(a,l,t), inCommit(i,...,l,t),
      dest(l',l,t), delay((i,...,a,l,t,l'),t')
```

On C', introduce the relation numPartitions(n) and populate with it the number of partition locations $addr_i$. Replace Lines 6 and 7 with the following code.

```
1   outReceived(i,count<...>,a,l,t) :- out(i,...,a,l,t)
2   # Sum the expected messages from all partitions.
3   outCountSum(i,sum<c>,a,l,t) :- outCount(p,i,c,a,l,t)
4   # Check if all partitions have sent their counts.
5   outCountPartitions(count<p>,i,a,l,t)
      :- outCount(p,i,c,a,l,t)
6   sealed(a,l,t)
      :- outReceived(i,c,a,l,t), outCountSum(i,c,a,l,t),
      outCountPartitions(n,i,a,l,t), numPartitions(n)
```

B.4.3 Proof. Unlike the proof in Appendix B.3.2, we cannot reuse the proof in Appendix B.1.2 because out is technically neither fully or partially partitioned; instead outCount is partially partitioned and the logic of C' is modified. Instead, we show that for instance I' after transformation, $r =$ sealedOut, for all time t, there exists I such that $I'_{r,t} = I_{r,t}$. The proof of correctness for all other relations in C is covered by Appendix B.3.2, as r is an output relation so no other relations in C are dependent on r.

Since r has an existence dependency on some input rSealed of C_1, we only need to consider the times when rSealed is not empty. Consider time t' in I' where rSealed contains some fact f'. By partial partitioning, there is no other fact in rSealed with the same time as f'. Consider all corresponding facts of f' for each partition p, and their corresponding times t'_p. By the proof in Appendix B.3.2, there exists a time t in I such that for each t'_p, $I'_{\overline{r'},p,t'_p} = I_{\overline{r},p,t}$.

Let φ be any rule in C with r in its head. The body of φ are referenced in C and included in $\overline{r'}$ in I' and $\overline{r}$ in I, respectively, and since those relations contain the same facts at times t'_p and t, the will evaluate to the same output facts f' and f, except the range of possible times for f' is (t'_p, ∞) while the range is (t, ∞) for f. By construction of I, $t'_p \geq t$, so the range of possible times for f' is a subset of the range for f, and any immediate consequence producing f' in I' is possible in I. Note that the addition of the body term inCommit does not affect the immediate consequence, since we assumed that r already has an existence dependency on in. Similar logic applies for the evaluation of outCount, completing our proof.

C NON-LINEARIZABLE EXECUTION OVER PARTITIONED ACCEPTORS

CScalablePaxos 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. Proposers in CScalablePaxos can become the leader after receiving replies from a quorum of any $f + 1$ acceptors for each set of n partitions; the partitions across quorums do not need to correspond to the same acceptors. In contrast, the n partitions of each acceptor in ScalablePaxos represent one original acceptor, so proposers in ScalablePaxos become the leader after receiving replies from all n partitions of a quorum of $f+1$ acceptors. Crucially, by allowing the highest ballot held at each partition to diverge, CScalablePaxos can introduce non-linearizable executions that remain safe for Paxos, but are too specific to generalize.

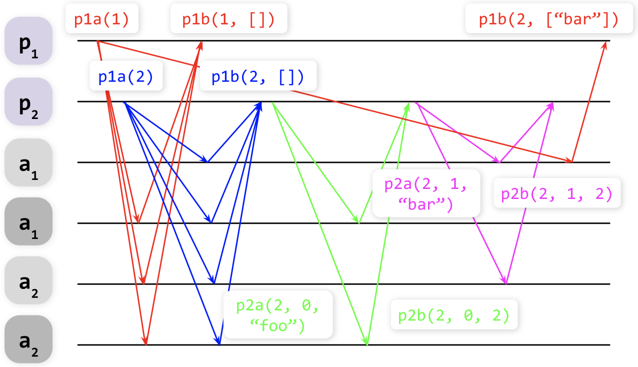


Figure 11: Non-linearizable execution of CScalablePaxos. Acceptor a_3 is excluded for simplicity. Lighter-gray acceptors belong to partition 1, and darker-gray acceptors to partition 2. Each set of requests and matching responses are a different color.

We first define what it means for a Paxos implementation to be linearizable. A p1a and its corresponding p1b correspond to a request and matching response in the history. For an implementation of Paxos to be linearizable, the content of each p1b must be consistent with its matching p1a taking effect some time between the p1a arrival time and the p1b response time. The same statements hold for p2a and matching p2b messages. Since p1a and p1b messages are now sent to each partition in CScalablePaxos, we must modify the definition of linearizability for CScalablePaxos accordingly. Assume that a p1a arriving at acceptor a in Paxos corresponds to the arrival of the same p1a messages at all partitions of a in CScalablePaxos, and a matching p1b arriving at proposer p in Paxos corresponds to the arrival of all matching p1b messages at p in CScalablePaxos.

Now consider the execution shown in Figure 11:

(1) Proposer p_1 broadcasts p1a with ballot 1. It arrives all acceptors except partition 1 of acceptor a_1. The other acceptors return p1b with ballot 1.
(2) Proposer p_2 broadcasts p1a with ballot 2. It arrives at all partitions of every acceptor, which return p1b with ballot 2. Proposer p_2 is elected leader.
(3) Proposer p_2 sends p2a with ballot 2, message “foo”, and slot 0 to partition 2 of every acceptor, which return p2b with ballot 2. “foo” is committed.
(4) Proposer p_2 then sends p2a with ballot 2, message “bar”, and slot 1 to partition 1 of every acceptor, which return p2b with ballot 2. “bar” is committed.
(5) Proposer p_1’s p1a finally arrives at partition 1 of a_1, which returns p1b with ballot 2, containing “bar” in its log. Proposer p_1 merges the p1bs it has received and concludes that the log contains only “bar”.

The execution is non-linearizable: by reading “bar”, p_1’s p1b must happen-after the write of “bar”, which happens-after the write of “foo”, but p_1 does not read “foo”, so it must happen-before the write of “foo”. Why is CScalablePaxos correct despite allowing such non-linearizable executions?

Non-linearizable reads of the log are only possible in Paxos when a proposer fails leader election, in which case, *the log is discarded* and the proposer tries again. The non-linearizable log is never used. Intuitively, because phase 1 (leader election) quorums must intersect, in order for a proposer p_1 to read a write from p_2 that occurred while p_1 was attempting leader election, p_2 must have completed leader election, overlapping in at least 1 acceptor with p_1 and preempting p_1. Thus p_1 will fail to become the leader and the log it receives in p1b does not matter.

These differences stem from optimizations that are specific to Paxos and require an in-depth, global understanding of the protocol. By design, our systematic optimization framework is protocol-agnostic and considers only local rewrites. In contrast, CScalablePaxos is able to admit optimizations that introduce non-linearizable executions as it can prove that the results of these executions are never used.