Serializability for Eventual Consistency: Criterion, Analysis, and Applications

Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev
Department of Computer Science, ETH Zurich, Switzerland
{lucas.brutschy, dimitar.dimitrov, peter.mueller, martin.vechev}@inf.ethz.ch

Abstract

Developing and reasoning about systems using eventually consistent data stores is a difficult challenge due to the presence of unexpected behaviors that do not occur under sequential consistency. A fundamental problem in this setting is to identify a correctness criterion that precisely captures intended application behaviors yet is generic enough to be applicable to a wide range of applications.

In this paper, we present such a criterion. More precisely, we generalize conflict serializability to the setting of eventual consistency. Our generalization is based on a novel dependency model that incorporates two powerful algebraic properties: commutativity and absorption. These properties enable precise reasoning about programs that employ high-level replicated data types, common in modern systems. To apply our criterion in practice, we also developed a dynamic analysis algorithm and a tool that checks whether a given program execution is serializable.

We performed a thorough experimental evaluation on two real-world use cases: debugging cloud-backed mobile applications and implementing clients of a popular eventually consistent key-value store. The experimental results indicate that our criterion reveals harmful synchronization problems in applications, is more effective at finding them than prior approaches, and can be used for the development of practical, eventually consistent applications.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; C.2.4 [*Computer-Communication Networks*]: Distributed Systems — Distributed databases

Keywords Replication, eventual consistency, serializability

1. Introduction

Modern distributed systems increasingly rely on replicated data stores [12, 19, 20, 33] in order to achieve high scalability and availability. As dictated by the CAP theorem [16], consistency, availability and partition-tolerance cannot be achieved at the same time. While various trade-offs exist, most replicated stores tend to provide relaxed correctness notions that are variants of eventual consistency: updates are not immediately but eventually propagated to other replicas, and replicas observing the same set of operations reflect the same state.

However, relaxations of strong consistency come at a price as applications may now experience unexpected behaviors not possible under strong consistency. These behaviors may lead to serious errors, and make development of such applications more challenging because the application itself is now responsible for guaranteeing strong consistency where required. Serializability then becomes an important criterion for reasoning about the correctness of the application: if serializability holds, one can reason about the application without considering the effects of weak consistency. Further, serializability violations can guide the placement of correct synchronization. While desirable, the general notion of serializability is very difficult to use in practice. A key issue is that deciding whether a concurrent execution is serializable is NP-hard [28]. This is one of the key reasons why stronger but computationally tractable serializability criteria, such as conflict serializability [28], have been explored (see also [4]).

Key Properties To be practically useful, a serializability criterion for eventual consistency must possess at least three key properties: (a) it must be *strong enough* so that the associated decision problem is computationally feasible, (b) it must be *precise enough* so it does not rule out desired application behaviors, and (c) it should handle the *weak semantics* of modern data stores.

Designing a serializability criterion that addresses all three properties is very challenging. For example, general serializability satisfies conditions (b) and (c), but not condition (a). On the other hand, conflict serializability satisfies condition (a), but neither (b) nor (c). Subsequent developments on conflict serializability improved on (b) and (c) but, interestingly, not on both at the same time. For example, the work of Weihl [38] improves the precision of conflict serializability under strong consistency by reasoning about commuting high-level operations. Other works, for instance [3, 14], incorporate weaker semantics, such as snapshot isolation and causal consistency, but do not improve the precision and still work with low-level reads and writes.

Key Challenge The main challenge then is: can we obtain a criterion that covers all three properties above: is computationally feasible, is precise enough to be used for practical applications, and can deal with (very) weak semantics such as eventual consistency?

This Work To addresses the above challenge, we propose a new serializability criterion that generalizes conflict serializability to eventually consistent semantics, while at the same time taking into account high-level operations. This enables precise reasoning about replicated data types (such as replicated maps and lists [9, 30]), which are commonly used in modern distributed applications. Since we assume only eventual consistency, our criterion immediately applies to all consistency levels that strengthen eventual consistency in various ways [9, 10, 23, 34].

The key technical insight of our work is that a precise criterion for eventual consistency needs to take into account not only commutativity, but also that some operations absorb (mask) the effects of others. Absorption is one of the main properties that permits an execution possible under eventual consistency to be recognized as equivalent to a strongly consistent one. The core technical problem here is finding a way to combine commutativity and absorption reasoning: a combination that is natural for reads and writes but non-trivial for abstract operations with rich semantics.

We note that in practice, a serializability criterion need not (and typically should not) be used on the entire application. It is most effective when used in a targeted manner for program parts intended to be serializable (e.g., payment check-out), while not used for parts that can tolerate weak consistency (e.g., display code) and can benefit from higher performance. Indeed, in our evaluation, we used our criterion in such a targeted way. This usage scenario is similar to how standard conflict serializability is used for shared memory concurrent programming (e.g., [37]).

To substantiate the usefulness of our criterion, we built a dynamic analyzer that checks whether the criterion holds on program executions, and evaluated our analyzer on two application domains. First, we analyzed 33 small mobile apps written in TOUCHDEVELOP [35], a framework that uses weakly consistent cloud types [9]. The experimental results indicate that our serializability criterion is able to detect violations that lead to some difficult-to-catch errors. Second, we implemented the database benchmark TPC-C [36] using the eventually consistent data store RIAK [19] and show how our criterion can guide developers to derive correct synchronization for client implementations.

Contributions The main contributions of our paper are:

- An effective serializability criterion for clients of eventually consistent data stores. Our criterion generalizes conflict serializability to deal with weakly consistent behaviors and high-level data types.
- Polynomial-time algorithms to check whether the criterion holds on a given program execution.
- An implementation of our algorithms for two data stores: the TOUCHDEVELOP cloud platform for mobile device applications and the distributed database RIAK.
- A detailed evaluation that indicates that our criterion is useful for finding previously undetected errors and can help in building correct and scalable applications running on eventually consistent data stores.

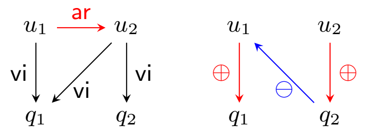
Outline In the next section, we provide an overview of the main concepts introduced in this paper and discuss how they relate to previous work. Section 3 describes our formal system model, and Section 4 presents our main result, a new serializability criterion. On this basis, Section 5 develops a dynamic detection algorithm, which is then evaluated in two practical settings in Section 6 and Section 7. Section 8 discusses related work and Section 9 concludes.

2. Overview

In this section, we provide an informal overview of the key challenges and illustrate our solution to these. Full formal details are presented in later sections.

Motivating Example Consider the following code fragment, adapted from a mobile gaming library[1]

```
Players.at(G, I).user.setIfEmpty(userID)
if Players.at(G, I).user != userID
  // try next position
```



(a) Visibility, arbitration

(b) Dependency, anti-dependency

Figure 1: Execution of motivating example, with operations:
u_1 `map(G,I).user.setIfEmpty("Alice")`
u_2 `map(G,I).user.setIfEmpty("Bob")`
q_1 `map(G,I).user` returns `"Alice"`
q_2 `map(G,I).user` returns `"Bob"`

Here, `Players` is a distributed map from a game `G` and a position `I` to a user participating in the game. A user identifier is stored in field `user`. For now, suppose that each operation such as `setIfEmpty` runs atomically in its own transaction (our results extend to transactions with multiple operations). The intended behavior is that all players have a consistent view of the `Players` map and, in particular, that it is not possible for two players to assume they have obtained the same position in a game. This behavior is guaranteed for executions under strong consistency: if two competing accesses to the same position are performed concurrently, one of the `setIfEmpty` operations will remain without effect, and the corresponding client has to try the next position in the game.

To illustrate the issues with eventual consistency, suppose we have two users, ‘Alice’ and ‘Bob’, trying to acquire the same position `I` in game `G`. Here, each user executes a `setIfEmpty` update, which will eventually propagate to the replica of each user, followed by a query on the map (executed on the user’s own replica). That is, as shown in Figure 1, ‘Alice’ performs update u_1 and query q_1, while ‘Bob’ performs update u_2 and query q_2. The figure also shows a possible execution of these updates and queries. In the graph, vi designates whether an update is visible to a query (i.e., whether the update was applied to the replica on which the query is executed before the query took effect). By ar, we denote an arbitration order in which all (conflicting) updates are ordered by the system. A query therefore observes a database state that is the result of applying such *visible* updates in the arbitration order.

Returning to our example in Figure 1a, here, both u_1 and u_2 are visible to q_1, but only u_2 is visible to q_2. Since u_1 is ordered before u_2 by arbitration, q_1 will read the value written by u_1 (u_2 will not override the value since it is a `setIfEmpty` operation), while q_2 will read the value written by the only visible update u_2. Consequently, both clients conclude that they have acquired the position in the game. The resulting behavior is not serializable: there is no sequential execution of u_1, q_1, u_2 and q_2 in which the queries q_1 and q_2 return these values.

In this work, we use the classic notions of dependency and anti-dependency [1] to characterize serializability: intuitively, a query *depends* on a visible update in the execution if the result of the query would change had the update become invisible. In Figure 1b, dependencies are marked with a $\oplus$ arc from the update to the query. Similarly, a query *anti-depends* on an update that is *not* visible if the query result would change had the update become visible. Figure 1b shows anti-dependencies as arcs labeled $\ominus$ from the query to the update. Here, q_2 anti-depends on u_1, because u_1 is ordered before u_2 in the arbitration order and, by the semantics of `setIfEmpty`, q_2 would therefore observe a different value if u_1 had become visible. If the four relations of dependency, anti-dependency, program order po, and arbitration order ar form a cycle then the execution is *not serializable* [14]. Indeed, in our example, when we put all these four relations together, we do have a cycle, namely: u_1, u_2, and q_2.

[1] “cloud game lobby”, written in TOUCHDEVELOP and available at `http://touchdevelop.com`

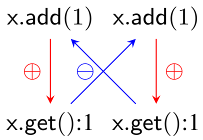


Figure 2: A trivial serializability violation with two counters

Commutativity and Absorption A precise serializability criterion requires a precise notion of dependency. For reads and writes, this is fairly straightforward: a read depends on the last-arbitrated write that is visible to it and accesses the same record. For operations on high-level data types such as counters, maps, sets, and tables, however, this is not as straightforward. Consider for instance the case where there is a second function in the above gaming library that lists all games a player with a given `userID` participates in:

```
Games := Players.select(_.user == userID)
```

Suppose user 'Alice' reserves a position in a game via the update `setIfEmpty("Alice")`, and concurrently 'Bob' lists the games that he participates in via the above query. Here, Alice's update is not a dependency for Bob's query as no matter whether it is visible or not, the result of the query will be the same: Alice's reservation does not influence Bob's participation. The reason is that, even though both operations access the same data, they actually *commute*. Thus, determining dependencies precisely requires reasoning about commutativity. In this work, we show how to leverage commutativity properties of arbitrary operations in order to capture the dependencies between them.

And while commutativity is useful, we can make our dependency definition even more precise. Consider again the execution shown in Figure 1b, but suppose that now we had used `set` instead of `setIfEmpty`. Even though both updates do not commute with either of the two queries, the execution is serializable in the order $u_1 q_1 u_2 q_2$. Indeed, we no longer have a cycle as u_2 hides the effect of u_1 from q_2, and therefore q_2 does not depend on u_1. We refer to this form of hiding as *absorption*. Again, if the operations are reads and writes, absorption is easy to define: updates absorb each other if they write to the same record. However, for the richer operations considered in this work (e.g., those of high-level data types), the definition of absorption may be more involved: operations such as `setIfEmpty` may be absorbing, non-absorbing, or absorbing only under specific conditions.

Targeted Serializability Checking Requiring serializability for all operations in a program is typically too strong in practice. Consider for example the execution in Figure 2, where a serializability violation occurs when a counter is incremented by two clients and queried by each of them without observing the increments from the other. Any application incrementing a counter and then querying its value without using synchronization is susceptible to such an anomaly, but it is often harmless. For example, an application might read the counter solely to display it to the user and there may be no negative effects of displaying an outdated value. To avoid reporting violations in such cases, we allow the developer to provide a lightweight specification of a set of operation invocations not subject to serializability. An execution is considered serializable with respect to this specification if it is serializable after projecting out the specified invocations. This is a slight generalization of the notion of atomic set serializability introduced by Vaziri et al. [37]. We discuss targeted serializability in more detail in Section 6.

Replicated Data Type Behavior Serializability can be used as a generic correctness criterion for an application, because the serial behaviors of a data type act as witnesses for the correctness of a weak execution. For example, consider an add-wins set [30], where conflicts between concurrent `add` and `remove` operations on the same element are resolved by considering the element to be added. The correctness of an application may then be witnessed by a serial execution, in which the `add` is executed after the `remove`. The behaviors of most replicated data types [6, 9, 18, 31] allow such serial witnesses. However, there are some exceptions to this rule. For example, the multi-value register of Dynamo [12] has the following semantics: if two writes to the register happen concurrently, then a later read observing the writes will retrieve *both* of the written values; otherwise, it will retrieve one of the values (the last one). In the first case, there cannot be a serial witness for the correctness of the application, as no serial execution of operations on the data type results in multiple values being read. If one is to reason about such data types, this means that one can no longer provide a default (serial) witness execution, but requires user input to specify which witnesses are acceptable. Thus, such settings are not a natural fit for serializability, but a different condition, which allows users to specify witnesses. Such manual intervention departs from the goal of the paper, which is to define a widely usable *generic* condition.

3. Weakly Consistent Systems

In this section, we present the model of weakly consistent systems that we later use to reason about serializability. Our model is formally equivalent to the one of Burckhardt et al. [8], but uses Mazurkiewicz traces instead of totally ordered event sequences. This choice makes reasoning about serializations more transparent, and is closer to the approach to weak memory by Shasha and Snir [32].

In the systems we consider, several processes interact with a weakly consistent data store, for example, a geo-replicated data store like Dynamo [12] or PNUTS [11]. Interaction between a process and the store happens in a sequence of atomic actions that manipulate or query the stored data. Our main subject of interest in such a system is the set of possible interaction histories that it can exhibit.

3.1 Actions

An action represents a primitive operation as issued by a process against the replicated data store. For example, storing 0 in a given record x is an action, denoted by x.set(0). Similarly, observing that x holds the value 0 is another action, denoted by x.get():0. Formally, we treat an *action* as a primitive operation whose arguments and results have been fixed to concrete values.

To make our arguments simpler, we assume that each action is either an update or a query. An *update* may modify the store but does not indicate a return value. On the other hand, a *query* must not modify the store but may return a value. Further, we assume that an update can always be applied, i.e., that it does not have any pre-condition. Our assumptions are non-restrictive as any action can be split into an query after an update, and any update can be made to do nothing if its pre-condition is not met.

We will model the weak semantics of the store based on the sequential semantics of actions. We assume that the sequential action semantics is given by a safety specification, that is, a prefix-closed set of finite action sequences, which we call *legal*. For example, the sequence x.set(0) x.get():0 is legal for standard read-write registers, while the sequence x.set(0) x.get():1 is not.

We can view an action sequence as a totally ordered set of *events* labeled by actions. Each action occurrence in the sequence corresponds to one event, e.g., x.set(0) x.set(0) x.get():1 consists of three events. We say that a given event $e \in \alpha$ is *legal* if the prefix of α ending in e is legal. Thus, the first two events in the last sequence are legal, while the third one is not.

Sometimes two action sequences α and β have the same behavior, that is, they read the same values and produce the same final state for any initial state. Then, we say that the two sequences are *equivalent*. Formally, this means that α and β are legal in the same contexts:

$$\alpha \equiv \beta \text{ iff } \forall \pi, \rho.\ \pi\alpha\rho \text{ is legal } \iff \pi\beta\rho \text{ is legal.}$$

With $\equiv$ we can easily state that two actions a, b *commute* ($ab \equiv ba$), or that one gets *absorbed* ($ab \equiv b$). For example:

$$\mathsf{x.set}(0)\ \mathsf{y.set}(1) \equiv \mathsf{y.set}(1)\ \mathsf{x.set}(0)$$

$$\mathsf{z.set}(0)\ \mathsf{z.set}(1) \equiv \mathsf{z.set}(1)$$

3.2 Traces

When reasoning about the possible sequences over a set of events, the order of commuting actions is usually irrelevant. It is easier to abstract it away and replace action sequences with what is known as traces (see, e.g., [26]). A trace relaxes the total order of a sequence to a partial one, preserving the order of non-commuting actions.

Given a partial order τ (or any binary relation), let $f \xrightarrow{\tau} g$ denote that τ relates f to g. A binary relation is *lower-finite* if each element is reachable along directed paths from at most finitely many others.

Definition 1. A *trace* is a lower-finite strict partial order τ on a countable set E of events such that for all $f, g \in E$:

(1.1) if $fg \not\equiv gf$ then $f \xrightarrow{\tau} g$ or $g \xrightarrow{\tau} f$, and
(1.2) if $fg \equiv gf$ and $f \xrightarrow{\tau} g$ then $f \xrightarrow{\tau} h \xrightarrow{\tau} g$ for some $h \in E$.

Condition (1.1) ensures that τ orders all pairs of non-commuting actions. This way, one can think of a trace as abstracting a set of equivalent sequences: this is the set L of all sequences over E that are consistent with the trace order. Indeed, starting with any $\alpha \in L$ we can obtain any $\beta \in L$ by swapping adjacent commuting actions.

Condition (1.2) ensures that τ makes no unnecessary ordering: applying it recursively to any two actions $f \xrightarrow{\tau} g$ such that $fg \equiv gf$, we will eventually obtain a sequence (by lower-finiteness)

$$f \xrightarrow{\tau} h_1 \xrightarrow{\tau} \dots \xrightarrow{\tau} h_n \xrightarrow{\tau} g,$$

where no pair of adjacent actions commute.

Below is an example of a trace where the record x gets read, then incremented twice, and finally read again:

$$\begin{tikzcd} & \mathsf{x.get}():0 \arrow[dl] \arrow[dr] & \\ \mathsf{x.add}(1) \arrow[dr] & & \mathsf{x.add}(2) \arrow[dl] \\ & \mathsf{x.get}():3 & \end{tikzcd} \tag{1}$$

Legality We define a trace to be *legal* iff all the action sequences it abstracts are legal. It is important to note that either all of these sequences are legal or none of them is, simply because they are all equivalent. Therefore, the trace is legal iff any of the sequences is legal, and respectively, illegal iff any of the sequences is illegal.

Note that if two actions a and b never appear adjacent in a legal sequence then they commute. The events $\mathsf{x.get}():0$ and $\mathsf{x.get}():3$ in (1) are an example. A trace that does not order such events admits illegal action sequences; consequently, such a trace itself is illegal and does not represent a sequential execution in the model. In legal traces, such events are always ordered.

Semi-traces In our proofs, we will need the trace analog of forming a subsequence. This is simply the *restriction* of the domain of a trace to a subset of its events. In this case, property (1.1) will still hold but (1.2) need not. We call such restrictions of traces *semi-traces*. For example, if we restrict the trace (1) to the two $\mathsf{get}()$ actions, then they remain ordered even though they commute:

$$\mathsf{x.get}():0 \rightarrow \mathsf{x.get}():3.$$

Concatenation Concatenation is an operation that applies to both traces and semi-traces. The concatenation of two semi-traces τ, υ over disjoint events is a semi-trace $\tau\upsilon$ over the union of those events. It is defined as the smallest partial order $\tau\upsilon \supseteq \tau \cup \upsilon$ such that

$$e_1 \in \tau, e_2 \in \upsilon, \text{ and } e_1e_2 \not\equiv e_2e_1 \implies e_1 \xrightarrow{\tau\upsilon} e_2.$$

For example, the trace (1) is the concatenation of two traces

$$\mathsf{x.get}():0 \rightarrow \mathsf{x.add}(1) \quad \text{and} \quad \mathsf{x.add}(2) \rightarrow \mathsf{x.get}():3.$$

Via concatenation, the most important notions on sequences lift to semi-traces: prefixes, legality of events, equivalence, etc.

3.3 Histories and Schedules

A history provides an external view on the events observed by each process in the system. We model a *process* as a possibly infinite sequence of events. Every process itself is split into *transactions*: contiguous segments that are intended to execute atomically. No transaction encompasses more than a single process.

Definition 2. A *history* (E, po, T) consists of

- a countable set E of events (each labeled by an action),
- a partial ordering po of E into a disjoint union of processes,
- a partition T of the processes into transactions $t_1, t_2, \dots \subseteq E$.

The connected components of po form the processes of the history. Moreover, because each process is lower-finite by definition, the whole po is lower-finite as well. Also, each transaction $t \in T$ resides on a single process, and therefore po orders t linearly.

The set of possible histories of the system defines its externally observable behaviors. To prevent undesired behaviors, a data store imposes constraints on this set, typically by requiring that each history can be scheduled in a specific way. A standard choice is to permit only *serializable* histories, i.e., those having serial schedules:

Definition 3. A *serial schedule* of a history (E, po, T) is a linear ordering so of E such that:

(3.1) the union $\mathsf{po} \cup \mathsf{so}$ is lower-finite and acyclic,
(3.2) every prefix of so is legal, and
(3.3) no two transactions $t_1 \neq t_2 \in T$ overlap, i.e., either:
(a) $f \xrightarrow{\mathsf{so}} g$ for all $f \in t_1$, $g \in t_2$, or
(b) $g \xrightarrow{\mathsf{so}} f$ for all $f \in t_1$, $g \in t_2$.

Serializability formalizes transaction atomicity: one can assume that all transactions appear as indivisible units of execution. This simplifies reasoning about concurrent processes a lot, because as long as each individual transaction preserves the required data invariants, any serializable history will preserve them too.

However, serializability is typically too expensive to enforce in a replicated setting: the CAP theorem [16] implies that during a network partition a store cannot be both available for updates and ensure serializability. That is why some stores choose various forms of eventual consistency instead. They are cheaper to enforce but also provide much weaker guarantees to the store clients. Consequently, clients need to implement their own synchronization between processes in order to ensure correctness.

In the present work, we consider the so-called strong eventual consistency[2] [31]. Informally, it is a combination of two properties: first, every process observes a consistent view, but only of a subset of the updates in the system so far; second, every update eventually propagates to the view of every process. We say that a history is *eventually consistent* iff it has a schedule with these properties:

[2] The motivation behind strong eventual consistency is that it captures the guarantees provided by most data stores better than broader forms of eventual consistency. This in turn enables more precise reasoning about client correctness. See [8] for a further discussion.

Relation	Type	Description
po	$E \times E$	process ordering
so	$E \times E$	serial ordering
vi	$U \times Q$	update visibility
ar	$U \times U$	update arbitration
$\oplus$	$U \times Q$	dependency
$\ominus$	$Q \times U$	anti-dependency

Table 1: The various model relations for a given set $E = U \cup Q$ of update events U and query events Q.

Definition 4. An *eventually consistent schedule* $(\mathsf{vi}, \mathsf{ar})$ of a history (E, po, T), with updates and queries $U \cup Q = E$, consists of

- a relation $\mathsf{vi} \subseteq U \times Q$ indicating the update-query visibility,
- a legal trace $\mathsf{ar} \subseteq U \times U$ arbitrating the order of updates in E,

such that they meet three conditions:

(4.1) the union $\mathsf{po} \cup \mathsf{vi}$ is lower-finite and acyclic,

(4.2) each query q is legal in the restriction of ar to $\{u \mid u \xrightarrow{\mathsf{vi}} q\}$,

(4.3) for any update u the set $\{q \mid u \overset{\mathsf{vi}}{\not\to} q\}$ is finite.

Eventual consistency imposes a weaker requirement on the legality of events than serializability (conditions (4.1) and (4.2)) and does not require transactions to be atomic.

Condition (4.1) is there to ensure consistency in the temporal sense: the transitive closure of $\mathsf{po} \cup \mathsf{vi}$ is a weakening of Lamport's *happened-before* relation [21]. If an event f is related to an event g by po or vi then f causally precedes g. Therefore, no pair of events should participate in a cycle of such causal relationships, nor should any event be causally preceded by infinitely many other events.

Condition (4.2) is there to ensure that each query q is consistent with the updates that it observes. This is the case if one obtains a legal semi-trace after concatenating the part of ar visible to q and the query q itself. Because ar is global for the whole schedule, it cannot be the case that one query considers a pair of updates ordered one way, while another query the opposite way.

Condition (4.3) is there to ensure that updates are eventually propagated. It says that an update is observed by almost all the queries, i.e., all but finitely many. This can happen only if the update gets delivered to all replicas eventually. Note that serial schedules have this property too: the only queries not observing a given update are those ordered before it, and these are finitely many.

There is an obvious way in which one can endow a serial schedule so with visibility and arbitration relations: a query observes all updates that so orders before it; two non-commuting updates get ordered the same way as so orders them. Thus, we will treat serial schedules as eventually consistent.

For brevity, we will omit "eventually consistent" and use the term *schedule*. Figure 1a shows a schedule.

Pre-schedules Later in Section 4.2, when we need to prove that a pair $(\mathsf{vi}, \mathsf{ar})$ forms a schedule, condition (4.1) will often hold trivially, while (4.2) and (4.3) will require more work. We call pairs $(\mathsf{vi}, \mathsf{ar})$ for which (4.1) holds *pre-schedules*. If condition (4.2) holds for a specific query q in a pre-schedule, then we will say that q is *legal in that pre-schedule*.

This concludes our model. Table 1 summarizes the notation that we introduced so far, plus two relations that we will consider next.

4. A Serializability Criterion

We will now present a sufficient criterion for checking a history's serializability when given one of its eventually consistent schedules. The main idea is to test whether the given schedule can be reordered into a serial one while preserving query legality. This is done by testing for acyclicity a certain directed graph, known as the dependency serialization graph, derived from the schedule.

The dependency serialization graph (see, e.g., [14]) is based on two relations between the events in a history: dependency and anti-dependency. Before stating our criterion we need to generalize this graph to our setting. The key step is to define dependency and anti-dependency for arbitrary actions, and not just for reads and writes as is done traditionally.

4.1 Dependency and Anti-dependency

We will first give suitable axioms for the notions of dependency and anti-dependency. The axiomatization decouples the correctness of the serializability criterion from the concrete instantiations of dependency and anti-dependency. In Sections 4.3 and 4.4, we will instantiate the axioms with concrete relations based on two algebraic properties of actions: commutativity and absorption.

4.1.1 Dependency

Informally, a query in a schedule depends on an update if this update is visible and potentially influences the query legality. For example, consider the serial schedule

$$\mathsf{x.add}(1)\ \mathsf{y.set}(1)\ \mathsf{x.set}(0)\ \mathsf{x.add}(2)\ \mathsf{x.get}():2\ \mathsf{y.get}():1 \qquad (2)$$

Here, $\mathsf{x.get}():2$ depends on both $\mathsf{x.set}(0)$ and $\mathsf{x.add}(2)$: if we remove any of them then the query becomes illegal for some initial states. However, if we remove the actions on y then the query will definitely remain legal. Thus, $\mathsf{x.get}():2$ depends only on $\mathsf{x.set}(0)$ and $\mathsf{x.add}(2)$. The situation with the query $\mathsf{y.get}():1$ is analogous: it depends only on $\mathsf{y.set}(1)$.

Instead of giving a single definition of when a query depends on a given update, we assume that for every pre-schedule $(\mathsf{vi}, \mathsf{ar})$ dependencies are specified as a *dependency relation* $\oplus \subseteq \mathsf{vi}$ so that

Axiom 1. *For every relation R such that $\oplus \subseteq R \subseteq \mathsf{vi}$, a query q is legal in $(\mathsf{vi}, \mathsf{ar})$ iff it is legal in the pre-schedule (R, ar).*

In other words, dependency $\oplus$ is a lower bound on relaxing visibility to some $R \subseteq \mathsf{vi}$ so that query legality with respect to R remains the same. In particular, $(\mathsf{vi}, \mathsf{ar})$ is legal iff (R, ar) is. Given a dependency relation, we say that q depends on u iff $(u, q) \in \oplus$.

Leaving the dependency relation as a parameter makes our criterion more widely applicable; one may, for instance, instantiate it with a dependency relation that is precise, one that is easy to compute, or one that reflects a particular action semantics. In Section 4.3, we will derive a general relation based on the commutativity and absorption properties of actions. In Section 5, we will discuss how to compute this general relation, but also how a stronger notion of absorption gives rise to a more efficient algorithm.

4.1.2 Anti-dependency

Anti-dependency is the natural counterpart of dependency. An update is an anti-dependency of a given query in a schedule if it is not visible to the query, but making it visible may change the query legality. Consider, for example, the serial schedule

$$\mathsf{x.add}(1)\ \mathsf{x.set}(0)\ \mathsf{x.add}(2)\ \mathsf{x.get}():2\ \mathsf{y.set}(1):1\ \mathsf{x.set}(1) \qquad (3)$$

Here, $\mathsf{x.set}(1)$ is an anti-dependency of the query $\mathsf{x.get}():2$ because if we make it visible by reordering it right before the query, then the query becomes illegal. On the other hand, $\mathsf{y.set}(1)$ does not affect the legality on any action on x, and therefore it is not an anti-dependency of $\mathsf{x.get}():2$.

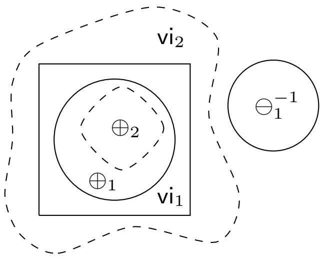


Figure 3: Inclusion relationship between visibility, dependency, and anti-dependency, given that vi_2 extends vi_1 over the same arbitration.

Similarly to dependency, we assume that an *anti-dependency relation* $\ominus \subseteq Q \times U$ is specified for every pre-schedule $(\mathsf{vi}, \mathsf{ar})$, where the converse $\ominus^{-1}$ is a subset of the complement of vi. However, instead of imposing a condition on legality directly, we require that anti-dependency is compatible with dependency:

Axiom 2. *If* $(\mathsf{vi}_1, \mathsf{ar})$ *and* $(\mathsf{vi}_2, \mathsf{ar})$ *are two pre-schedules such that* $\mathsf{vi}_1 \subseteq \mathsf{vi}_2$*, and* $\ominus_1^{-1} \cap \mathsf{vi}_2 = \varnothing$ *then* $\oplus_1 \supseteq \oplus_2$.

The axiom postulates that if visibility gets extended without making anti-dependencies visible then no new dependencies are introduced, as illustrated in Figure 3. By Axiom 1, this requirement is sufficient to guarantee that the two pre-schedules $(\mathsf{vi}_1, \mathsf{ar})$ and $(\mathsf{vi}_2, \mathsf{ar})$ have exactly the same legal queries.

4.2 The Criterion

In order to check whether a history is serializable, we assume that one of its eventually consistent schedules $(\mathsf{vi}, \mathsf{ar})$ is given, and use it to establish a witness schedule so for serializability. This serial witness is not arbitrary, but we require that it satisfies the following inequality over the relations of the given schedule:

$$\mathsf{so} \supseteq \mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus \tag{4}$$

Including po in so is necessary to ensure (3.1), while including the other three is a choice that makes it sufficient to show that so is a *pre-schedule*; its legality then follows automatically from the legality of $(\mathsf{vi}, \mathsf{ar})$. Note that we do not require $\mathsf{so} \supseteq \mathsf{vi}$, as it is not needed for serializability.

To check whether such a serial pre-schedule exists, we observe that (4) is equivalent to a set of ordering constraints on the history's transactions. We express these constraints as a graph:

Definition 5. The *dependency serialization graph* (DSG) of a given pre-schedule $(\mathsf{vi}, \mathsf{ar})$ of a history (E, po, T) is a directed graph whose nodes are the transactions in T, and which contains an arc (s, t) iff $s \neq t$ and $\mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus$ relates an event $f \in s$ to an event $g \in t$.

Every solution of inequality (4) implies that the dependency serialization graph is acyclic. We will prove the converse, namely, that acyclicity implies the existence of a solution. This way we reduce serializability testing to detecting cycles in a directed graph:

Theorem 1. *A history* (E, po, T) *with a finite number of processes is serializable if it has a schedule* $(\mathsf{vi}, \mathsf{ar})$ *with an acyclic DSG.*

Proof. Suppose that the DSG is acyclic and that it has a lower-finite topological ordering. By Definition 5, this ordering corresponds to a serial pre-schedule $\mathsf{so} \supseteq \mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus$. Let us denote its visibility relation with $\mathsf{vi}_{\mathsf{so}}$. The intersection $\mathsf{vi}_\cap = \mathsf{vi} \cap \mathsf{vi}_{\mathsf{so}}$ satisfies the condition $\oplus \subseteq \mathsf{vi}_\cap \subseteq \mathsf{vi}$ of Axiom 1, and therefore every query in E is legal in the pre-schedule $(\mathsf{vi}_\cap, \mathsf{ar})$. Because $\mathsf{vi}_\cap \subseteq \mathsf{vi}_{\mathsf{so}}$, this pre-schedule and $(\mathsf{vi}_{\mathsf{so}}, \mathsf{ar})$ satisfy the condition of Axiom 2, thus $\oplus_{\mathsf{so}} \subseteq \mathsf{vi}_\cap \subseteq \mathsf{vi}_{\mathsf{so}}$. Applying Axiom 1 to $(\mathsf{vi}_{\mathsf{so}}, \mathsf{ar})$, we conclude that the pre-schedule so is indeed a schedule.

We still need to establish that the serial pre-schedule so exists. Because the DSG is acyclic and there is a finite number of processes, we could use a round-robin scheduler to produce it. But for that, we need to prove that the DSG is lower-finite.

First, observe that no infinite path of the DSG can have a final node u_0, that is, no path is of the shape $\cdots \to t_{-2} \to t_{-1} \to t_0$. This is because such a path visits infinitely many transactions of at least one process (as there are only finitely many processes). However, by the definition of a history, every process is lower-finite, and a final node t_0 in the path implies that only finitely many transactions were visited.

Second, the schedule $(\mathsf{vi}, \mathsf{ar})$ is eventually consistent, and so all the relations po, ar, $\oplus$, and $\ominus$ are lower-finite (anti-dependency $\ominus$ is lower-finite because of eventuality (4.3)). It follows that every transaction has only a finite number of immediate predecessors, i.e., that the graph has a finite fan-in. Combined with the "no infinite final paths" property, we conclude that the DSG is lower-finite. □

Note that eventual consistency is not a safety property as (4.3) is a liveness condition and makes sense only for infinite histories. That is why we consider infinite histories in the first place. Curiously enough, serializability is not a safety property either: one can easily construct an example where all finite prefixes of a history are serializable, but the whole history is not (say, because it has no schedule satisfying (4.3)). On the other hand, our acyclicity criterion is a safety property: if violated by a history then it is violated in some finite prefix of it. Therefore, we have a classic case of under-approximating a non-safety property with a safety one.

So far we have shown the correctness of our criterion for all dependency and anti-dependency relations that satisfy Axiom 1 and Axiom 2. In the next subsections, we present concrete relations and show that they indeed have the required properties.

4.3 Commutativity, Absorption, Dependency

While it is possible to specify dependency and anti-dependency relations manually, practical applicability calls for an automatic construction. In this subsection, we propose a dependency relation that is based on two algebraic properties of actions f, g:

$$f \text{ commutes with } g \iff fg \equiv gf$$
$$f \text{ is absorbed by } g \iff fg \equiv g$$

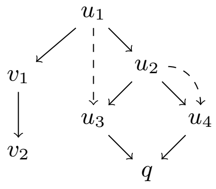
We consider an update u to be a dependency of a query q unless we can repeatedly apply commutativity and absorption as rewrite rules to obtain a schedule where u is not visible to q. We do not perform a commutativity rewrite after an absorption rewrite. This restriction simplifies the rewriting process at the cost of some generality (i.e., we may flag more actions dependent than optimal).

As an example, consider again the serial schedule (2) from Section 4.1. There, $\mathsf{y.set}(1)$ commutes with all updates on x. Thus, it is not a dependency of the query $\mathsf{x.get}()$:2, because we can swap commuting actions and obtain:

$$\mathsf{x.add}(1)\ \mathsf{x.set}(0)\ \mathsf{x.add}(2)\ \mathsf{x.get}():2\ \mathsf{y.set}(1)\ \mathsf{y.get}():1$$

In turn, here we can remove $\mathsf{x.add}(1)$ because $\mathsf{x.set}(0)$ absorbs it. After the removal, $\mathsf{x.add}(1)$ will not be visible to $\mathsf{x.get}()$:2, and therefore, is not a dependency of $\mathsf{x.get}()$:2.

Using commutativity and absorption to find the dependencies of a query q in an eventually consistent schedule $(\mathsf{vi}, \mathsf{ar})$ is a little bit more involved. We will rewrite the semi-trace induced by q and the updates that it observes. Namely, we restrict ar to the updates visible to q, and then append q itself. For example, such an induced semi-trace could be the following:



Here, solid arrows indicate the trace order, and dashed arrows indicate absorption. The two updates v_1 and v_2 are not dependencies of q because they do not precede it in the trace order, and therefore can be "moved past" q. The update u_2 is not a dependency of q either, as it gets absorbed by the adjacent update u_4. But after that, u_1 and u_3 become adjacent, and so u_1 gets absorbed by u_3. What remains are the two dependencies u_3 and u_4 of q.

We capture the above observations with two operations that remove non-dependencies from a given visibility relation: one for commutativity and one for absorption. With their help, we define the dependency relation of a pre-schedule:

Definition 6. For a given pre-schedule $(\mathsf{vi}, \mathsf{ar})$ we define two operations $\sharp$ and $\triangleright$ over relations $R \subseteq \mathsf{vi}$:

(6.1) $(u, q) \in R^\sharp$ iff $u \xrightarrow{R} q$ and there is an update $v \xrightarrow{R} q$ such that $vq \not\equiv qv$, and either $u \xrightarrow{\mathsf{ar}} v$ or $u = v$.

(6.2) $(u, q) \in R^\triangleright$ iff $u \xrightarrow{R} q$ and also for all updates $v \xrightarrow{\mathsf{vi}^\sharp} q$ if $u \xrightarrow{\mathsf{ar}} v$ but $u \xrightarrow{\mathsf{ar}} w \xrightarrow{\mathsf{ar}} v$ for no $w \xrightarrow{R} q$ then $uv \not\equiv v$.

The *dependency relation* of a pre-schedule $(\mathsf{vi}, \mathsf{ar})$ is the largest $\oplus \subseteq \mathsf{vi}^\sharp$ such that $\oplus = \oplus^\triangleright$.

The idea behind the definition is to shrink down the visibility vi to the dependency relation $\oplus$ by using commutativity and absorption, while preserving query legality. The $\sharp$ operation removes any events due to commutativity: if an update u and the updates arbitrated after it commute with a query q then they are not dependencies of q. Similarly, the $\triangleright$ operation removes updates that get absorbed by adjacent updates. Because absorbing an event may allow for others to be absorbed too (as in the example above), we look for the largest relation in which no further absorption is possible.

We will now prove that the relation $\oplus$ from Definition 6 is indeed a dependency relation, i.e., that it satisfies Axiom 1:

Theorem 2. *Given some pre-schedule* $(\mathsf{vi}, \mathsf{ar})$*, for every relation* R *such that* $\oplus \subseteq R \subseteq \mathsf{vi}$*, a query* q *is legal in* $(\mathsf{vi}, \mathsf{ar})$ *iff it is legal in the pre-schedule* (R, ar)*.*

Proof. To make our argument simpler, let us introduce some notation. We are looking at relations in the interval

$$[\oplus, \mathsf{vi}] = \{R \mid \oplus \subseteq R \subseteq \mathsf{vi}\}.$$

For brevity, let us collect all the restrictions of ar that such a relation R determines into a single map $\langle R \rangle$:

$$q \mapsto \mathsf{ar} \restriction \{u \mid u \xrightarrow{R} q\}.$$

Operations on semi-traces extend pointwisely to such maps, and if we denote the identity map $q \mapsto q$ with Q, then our claim becomes:

$$\forall R \in [\oplus, \mathsf{vi}].\, \langle R \rangle \cdot Q \text{ is legal} \iff \langle \oplus \rangle \cdot Q \text{ is legal}.$$

The basis of our proof is that $\oplus$ can be derived from any relation $R \in [\oplus, \mathsf{vi}]$ as the limit of the decreasing sequence:

$$R = R_0 \supseteq R_0 \cap \mathsf{vi}^\sharp = R_1 \supseteq R_1^\triangleright = R_2 \supseteq R_2^\triangleright = R_3 \supseteq \dots$$

This is a simple consequence of vi being lower-finite plus a standard fixed-point argument applied to the $\triangleright$ operation (see, e.g., [15, Proposition II-2.4]). From here we will show that every step i above preserves legality in the sense that

$$\langle R \rangle \cdot Q \text{ is legal} \iff \langle R_i \rangle \cdot Q \text{ is legal}.$$

That turns to be sufficient as the above property is preserved when taking limits, again a consequence of lower-finiteness.

So let us make the first step. The $\mathsf{vi}^\sharp$ relation splits $\langle R \rangle$ into two parts, the second one commuting with Q:

$$\langle R \rangle \cdot Q \equiv \langle R \cap \mathsf{vi}^\sharp \rangle \cdot \langle R \setminus \mathsf{vi}^\sharp \rangle \cdot Q \equiv \langle R \cap \mathsf{vi}^\sharp \rangle \cdot Q \cdot \langle R \setminus \mathsf{vi}^\sharp \rangle.$$

Since updates are free of pre-conditions, the right-most side is legal iff its prefix $\langle R \cap \mathsf{vi}^\sharp \rangle \cdot Q$ is legal.

Now, the remaining steps. It is enough to show that each restriction $\langle R_{i+1} \rangle$ is equivalent to $\langle R_i \rangle$. Let us first study how updates get absorbed when applying the $\triangleright$ operation. Recall definition (6.2), and consider the relation:

$$u \prec_i^q v \text{ iff } (u, q) \in R_i \setminus R_{i+1} \text{ breaks (6.2) because } v \xrightarrow{\mathsf{vi}^\sharp} q.$$

Because absorption is transitive, this relation is transitive in the sense that for all $i > j$:

$$u \prec_i^q v \prec_j^q w \implies u \prec_i^q w.$$

This allows us to conclude that if we remove (u, q) at step i, then it is always because of some v visible to q at that step:

$$(u, q) \in R_i \setminus R_{i+1} \implies u \prec_i^q v \xrightarrow{R_i} q \text{ for some } v.$$

Indeed, recursively tracing the reason for removals, we get a sequence of steps $i = i_n \geq i_{n-1} \geq \dots \geq i_1$ and updates:

$$u \prec_{i_n}^q v_n \prec_{i_{n-1}}^q \cdots \prec_{i_1}^q v_1 = v,$$

such that $v \xrightarrow{R_i} q$ (because otherwise, v was removed even earlier and we can continue the sequence).

We are now ready to prove that $\langle R_i \rangle \equiv \langle R_{i+1} \rangle$ for $i \geq 1$. By our previous argument, the updates removed from their relation R_i with q are the non-maximal vertices of the dag

$$(V = \{u \mid u \xrightarrow{R_i} q\}, E = \prec_i^q).$$

After sorting the non-maximal vertices $u_1, \dots, u_{n(q)}$ in V topologically, consider the sequence of trace restrictions:

$$\alpha_0 = \mathsf{ar} \restriction V \supset \alpha_1 = \alpha_0 \setminus u_1 \supset \cdots \supset \alpha_{n-1} \setminus u_{n(q)} = \alpha_n.$$

By definition (6.2), every α_j is of the form $\beta_j u_j v_j \gamma_j$, where $\beta_j v_j \gamma_j = \alpha_{j+1}$ and $u_j \prec_i^q v_j$. Here, v_j absorbs u_j, and therefore $\alpha_j \equiv \alpha_{j+1}$. We conclude that $\alpha_1 \equiv \alpha_{n(q)}$ for every query q, or in other words $\langle R_i \rangle \equiv \langle R_{i+1} \rangle$. □

4.4 A Corresponding Anti-dependency

We now match the dependency relation $\oplus$ defined in the previous subsection with a suitable anti-dependency relation.

Definition 7. For any pair $u \not\xrightarrow{\mathsf{vi}} q$ of an update and a query in a given pre-schedule $(\mathsf{vi}, \mathsf{ar})$, let $\oplus^{uq}$ denote the dependency relation with respect to the modified visibility $\mathsf{vi}^{uq} = \mathsf{vi} \cup \{(u, q)\}$. We define the *anti-dependency relation* of the pre-schedule as

$$q \xrightarrow{\ominus} u \text{ iff } u \xrightarrow{\oplus^{uq}} q.$$

In other words, we consider an update u to be an anti-dependency of a query q if it is not visible to q, but if making it visible would turn it into a dependency of q.

In Definition 7 we implicitly assume that $\oplus$ is the dependency relation from Definition 6, even though it might work for others too. We will next prove that $\ominus$ is indeed an anti-dependency relation, i.e., that it satisfies Axiom 2:

Theorem 3. *If* $(\mathsf{vi}_1, \mathsf{ar})$ *and* $(\mathsf{vi}_2, \mathsf{ar})$ *are two pre-schedules such that* $\mathsf{vi}_1 \subseteq \mathsf{vi}_2$*, and* $\ominus_1^{-1} \cap \mathsf{vi}_2 = \varnothing$ *then* $\oplus_1 \supseteq \oplus_2$.

Proof. Reasoning about commutativity is mostly straightforward, so we will focus on absorption here, i.e., assume $\mathsf{vi}_1 = \mathsf{vi}_1^\sharp$, and $\mathsf{vi}_2 = \mathsf{vi}_2^\sharp$.

Consider any two pre-schedules $(\mathsf{vi}_X, \mathsf{ar})$ and $(\mathsf{vi}_Y, \mathsf{ar})$, such that $\mathsf{vi}_X \subseteq \mathsf{vi}_Y$. The respective operations $\triangleright^X$ and $\triangleright^Y$ possess a kind of monotonicity property. For every pair of relations $A \in [\oplus_X, \mathsf{vi}_X]$ and $B \in [\oplus_Y, \mathsf{vi}_Y]$, update u, and a query q

$$A(u,q) \supseteq B(u,q) \implies A^{\triangleright^X}(u,q) \supseteq B^{\triangleright^Y}(u,q),$$

where $R(u,q)$ stands for the set $\{v \in R^{-1}(q) \mid u \xrightarrow{\mathsf{ar}} v \text{ or } u = v\}$. By the fixed-point argument from the proof of Theorem 2, this property transfers to the respective fixed-points:

$$A(u,q) \supseteq B(u,q) \implies \oplus_X(u,q) \supseteq \oplus_Y(u,q).$$

Moving on to the two pre-schedules $(\mathsf{vi}_1, \mathsf{ar})$ and $(\mathsf{vi}_2, \mathsf{ar})$, suppose that $(u,q) \in \mathsf{vi}_2$. As $\mathsf{vi}_1(v,p) \supseteq \oplus_1^{uq}(v,p)$ for every pair $(v,p) \in \mathsf{vi}_1$, we conclude that

$$\oplus_1(v,p) \supseteq \oplus_1^{uq}(v,p).$$

But $q \not\xrightarrow{\ominus_1} u$, i.e., $u \not\xrightarrow{\oplus_1^{uv}} q$, and therefore $\oplus_1 \supseteq \oplus_1^{uq}$. With this fact at hand, we will prove that for all $(u,q) \in \oplus_2$

$$\oplus_1(u,q) \supseteq \oplus_2(u,q).$$

Proceeding by well-founded induction on the set $\oplus_2(u,q)$, let $v \neq u$ belong to it. By the inductive hypothesis:

$$\oplus_1(u,v) \supseteq \oplus_1(v,q) \supseteq \oplus_2(v,q) \ni v.$$

We can, therefore, conclude that $\oplus_1(u,q) \setminus u \supseteq \oplus_2(u,q) \setminus u$. Now, if we let $R = \oplus_1 \cup (u,q)$, then we obtain:

$$R(u,q) \supseteq \oplus_2(u,q)$$

Because the relation R belongs to the interval $[\mathsf{vi}_1^{uq}, \oplus_1^{uq}]$, the monotonicity property applies here, and so:

$$\oplus_1(u,q) \supseteq \oplus_1^{uq}(u,q) \supseteq \oplus_2(u,q).$$

□

5. Detection Algorithm

In this section, we present two algorithms for detecting serializability violations given a history and a corresponding eventually consistent schedule. The first one is general, while the second makes assumptions on the data types, but is asymptotically more efficient. Detecting serializability violations amounts to determining the dependencies $\oplus$ and anti-dependencies $\ominus$ of the pre-schedule, and performing cycle detection. The latter has well-known linear-time solutions, and thus we discuss how to compute $\oplus$ and $\ominus$ in this section.

Our algorithms assume for each data type two specifications, a *commutativity specification* $\sharp$ on all pairs of actions and an *absorption specification* $\triangleright$ on all pairs of updates, which give sufficient conditions for commutativity and absorption:

$$u \sharp v \implies uv \equiv vu,$$
$$u \triangleright v \implies uv \equiv v$$

In practice, for each pair of operations, we provide a first-order logic formula that can be checked, given the arguments and return values of two actions, in time independent of the size of the graph. An example of a very simple absorption and commutativity specification for a dictionary is given in Figure 4.

	$\mathsf{put}_x[k', v']$	$\mathsf{get}_x[k', v']$	$\mathsf{size}_x[n']$
$\mathsf{put}_x[k, v]$	$k \neq k'$ or $v = v'$	$k \neq k'$	never
$\mathsf{get}_x[k, v]$	$k \neq k'$	always	always
$\mathsf{size}_x[n]$	never	always	always

(a) Commutativity specification

	$\mathsf{put}_x[k', v']$
$\mathsf{put}_x[k, v]$	$k = k'$

(b) Absorption specification

Figure 4: Specifications for a dictionary

5.1 Generic Algorithm

Algorithm 1 directly implements the mathematical construction given in Section 4.3 and Section 4.4. Here, PRUNE (line 12) takes a set of dependencies V and an arbitration order E between them, and computes the fixed-point $V^\triangleright$ using a standard work-list algorithm. In line 13, the arbitration order is transitively reduced. Then, in every step of the work-list computation in lines 14-19, a pair of updates u_1, u_2 is removed from the work-list and it is checked whether u_1 is absorbed by u_2, its successor in the transitively reduced arbitration order. If so, DELFROMREDUCTION in line 18 removes the update u_1 from (V, E), inserts edges from all predecessors to all successors of u_1, re-computes the transitive reduction, and returns all newly inserted edges. It thereby preserves both reachability and transitive reduction over vertex removals.

For the purpose of explaining the algorithm, we differentiate between *direct* (anti-)dependencies of a query q, which are non-commutative with q, and *indirect* (anti-)dependencies, which are commutative with q but are arbitrated before a direct dependency.

DEPENDENCIES uses PRUNE to compute both $\oplus$ and $\ominus$. For each query q, it first determines U_r, the set of all updates related to q by $\mathsf{vi}^\sharp$ in Definition 6. U_r therefore contains the set of all updates non-commutative with and visible to q, as well as all updates preceding them in the arbitration order. This set must necessarily include all direct or indirect dependencies, as well as indirect anti-dependencies. It then uses PRUNE to eliminate all absorbed updates in line 5, where we use $\mathsf{ar} \restriction U_r$ to denote the restriction of the relation ar to elements in U_r. All remaining updates are added as dependencies of q in line 6. In the second step (line 7 onwards), it re-inserts one invisible update after the other and checks whether it is absorbed by the dependencies of q. If not, it is added to the anti-dependencies. The complexity of the algorithm is $O(n^4 m)$, where $n = |U| + |Q|$ and $m = |\mathsf{ar}|$. The polynomial complexity shows that our criterion is strong enough to make the checking computationally feasible.

5.2 Optimized Algorithm

The complexity of the generic algorithm is caused largely by the fact that absorption can, in general, be used to prune dependencies only if the absorbed update is directly followed by the absorbing update in the arbitration order. For example, assume a store provides a swap operation that atomically swaps two fields of a record. We can conclude that two writes to a field of a record absorb each other only if there is no swap update on the written field in between the two field writes. Generally, we can prune an update only if it has an edge to an absorbing update in the transitive reduction of the arbitration order, which forces us to recompute the transitive reduction after every prune operation.

We can strengthen the notion of absorption to be agnostic to updates arbitrated between the absorbed and the absorbing updates. We call this *far-reaching* absorption. For data types with operations for which far-reaching absorption differs from standard absorption

Algorithm 1 Generic algorithm for determining the dependencies and anti-dependencies for a schedule with updates U, queries Q, visibility vi and arbitration ar

```
 1: function DEPENDENCIES($U, Q, \mathsf{vi}, \mathsf{ar}$)
 2:     $(\oplus, \ominus) \leftarrow (\emptyset, \emptyset)$
 3:     for $q \in Q$ do
 4:         $U_r \leftarrow \{u \in U \mid (u, q) \in \mathsf{vi}^\sharp\}$
 5:         $U_p \leftarrow$ PRUNE $(U_r, \mathsf{ar} \upharpoonright U_r)$
 6:         $\oplus \leftarrow \oplus \cup (U_p \times \{q\})$
 7:         for $u \in \{u \in U \mid u \not\xrightarrow{\mathsf{vi}} q\}$ do
 8:             $U_u \leftarrow$ PRUNE $(U_p \cup \{u\}, \mathsf{ar} \upharpoonright (U_p \cup \{u\}))$
 9:             if $u \in U_u$ then
10:                 $\ominus \leftarrow \ominus \cup \{(q, u)\}$
11:     return $(\oplus, \ominus)$

12: function PRUNE($V, E$)
13:     $E \leftarrow$ TRANSITIVEREDUCTION($E$)
14:     $W \leftarrow E$
15:     while $W \neq \emptyset$ do
16:         $((u_1, u_2), W) \leftarrow$ REMOVEELEMENT($W$)
17:         if $u_1 \rhd u_2$ then
18:             $(V, E, N) \leftarrow$ DELFROMREDUCTION($V, E, u_1$)
19:             $W \leftarrow E \cap (W \cup N)$
20:     return $V$
```

(such as swap), this strengthening will result in more dependencies and therefore a less precise serializability criterion. However, many systems, including the two discussed in the next two sections, do not contain such data types. Here we can employ the strengthened notion and an algorithm based on it without any loss of precision.

Formally, we have far-reaching absorption between two updates u, v, denoted by $u \blacktriangleright v$, if and only if for all traces of updates χ, we have $u\chi v \equiv \chi v$. For data stores for which we have $\blacktriangleright = \rhd$, the algorithm we are about to present is as precise as the generic algorithm.

The optimized Algorithm 2 computes U_r, the set of all updates that may form direct or indirect dependencies, or indirect anti-dependencies, as in the generic version. We then check for all other updates whether they are invisible and non-commutative with q, and thus form direct anti-dependencies. In the second step (line 8 onwards), all updates absorbed by q-visible successors in the arbitration order are eliminated. Depending on them being visible to q or not, the remaining elements of the set are added as dependencies or anti-dependencies, resp. (line 11 onwards). The complexity of the algorithm is $O(\max(nm, n^2))$, where $n = |U| + |Q|$ and $m = |\mathsf{ar}|$, and thereby significantly faster than the generic version.

6. Application: Debugging Cloud-Backed Mobile Software

In this section, we describe and evaluate a dynamic analysis tool for checking serializability violations in TOUCHDEVELOP [35] applications and show that our criterion is precise enough such that violations are likely to indicate actual bugs.

TOUCHDEVELOP is a platform for mobile device applications providing direct integration of replicated cloud-backed storage. We compare our results to the notion of commutativity races [13] and show that our criterion is better suited for debugging, as it captures harmful violations more precisely: over all applications, our criterion flags 75% less potential serializability violations.

First, we describe the TOUCHDEVELOP system briefly. Then, we discuss a prototype implementation of our tool ECRACER. Finally, we discuss the serializability violations found.

Algorithm 2 Optimized algorithm for determining the dependencies and anti-dependencies for a schedule with updates U, queries Q, visibility vi, and arbitration ar

```
 1: function FASTDEPENDENCIES($U, Q, \mathsf{vi}, \mathsf{ar}$)
 2:     $(\oplus, \ominus) \leftarrow (\emptyset, \emptyset)$
 3:     for $q \in Q$ do
 4:         $U_r \leftarrow \{u \in U \mid (u, q) \in \mathsf{vi}^\sharp\}$
 5:         for $u \in (U \setminus U_r)$ do
 6:             if $u \not\xrightarrow{\mathsf{vi}} q \wedge u \not\sharp\, q$ then
 7:                 $\ominus \leftarrow \ominus \cup \{(q, u)\}$
 8:         for $u, u' \in U_r$ with $(u, u') \in \mathsf{ar}$ do
 9:             if $u \blacktriangleright u' \wedge u' \xrightarrow{\mathsf{vi}} q$ then
10:                 $U_r \leftarrow U_r \setminus \{u\}$
11:         for $u \in U_r$ do
12:             if $u \xrightarrow{\mathsf{vi}} q$ then
13:                 $\oplus \leftarrow \oplus \cup \{(u, q)\}$
14:             else
15:                 $\ominus \leftarrow \ominus \cup \{(q, u)\}$
16:     return $(\oplus, \ominus)$
```

While we focus on a relatively narrow type of applications targeting a specific system, the ideas in this section are generally applicable to a large class of so-called causally consistent data stores [9, 23, 24, 29]. We will discuss an analysis for an eventually consistent key-value store in the next section.

6.1 Cloud Types

TOUCHDEVELOP uses the global sequence protocol [9] to implement a replication system providing *prefix consistency*. In a prefix-consistent system, a client observes a prefix of a common global sequence of updates, plus its own updates after the end of the prefix. This property is stronger than causal consistency but weaker than snapshot isolation [10]. All three are stronger than eventual consistency and therefore our criterion can be used directly.

All TOUCHDEVELOP code executes within weak transactions that provide *atomic visibility*, that is, they guarantee a stable view of a prefix-consistent snapshot of the data store. Updates propagate asynchronously to other clients at the end of each transaction. Transaction boundaries are inserted whenever the runtime is idle, e.g., between the execution of event handlers or during execution of blocking operations.

The replication system is exposed to the programmer as *cloud types*: data types that behave similarly to regular heap-stored data structures, but are replicated automatically to other clients. Cloud types include high-level data structures such as maps and lists, but also simple data types with a richer set of atomic operations. For example, a cloud integer can be set to a certain value using set, but also supports a commutative add operation.

To synchronize, clients can query whether their last update on a cloud type is *confirmed*, meaning that the update was included in the global prefix, and all updates that precede it in the prefix are visible to the client.

6.2 Prototype Implementation

Our tool ECRACER performs dynamic offline serializability analysis based on the optimized algorithm in Section 5.2. First, the TOUCHDEVELOP client runtime is instrumented to record the execution history and schedule of a client program. Second, an analysis back-end reads the recorded information from an execution with two or more clients and detects serializability violations as discussed in the previous sections. The violations, which are embodied by cycles in dependency serialization graphs (DSGs), are then mapped back to source code locations and reported to the user.

Recording The instrumentation of the TOUCHDEVELOP runtime records events and stores them locally. To reconstruct the visibility relation vi between events in the system, we replicate vector clocks [25] using the data store of TOUCHDEVELOP itself. That is, we keep a replicated map from client identifiers to integer counters, and every update to replicated data is instrumented with an update to the client's logical clock. This yields correct vector clocks, since (a) atomic visibility guarantees that a client will observe the counter increment if and only if it observes the corresponding update, and (b) causal consistency guarantees that all increments causally preceding an update will be observed. The overhead of the instrumentation is low, as a single increment will only result in a small constant number of extra bytes being sent to the other replicas. The arbitration order ar is not recorded, as it is only known by the server and not communicated to the client. Note, however, that Theorem 1 does not require us to use the real schedule when checking for serializability, but any schedule in which all queries of the history are legal. We can, therefore, assume arbitration by physical time of the client and check legality of the resulting schedule. In our experiments, this approach yielded legal schedules in all cases.

Analysis The analysis back-end implements the optimized algorithm from Section 5.2, instantiated with commutativity and absorption specifications of all operations in TOUCHDEVELOP. For our experiment, the boundaries of intended transactions (T in Definition 2) coincide with the boundaries of the above mentioned weak transactions.

As explained in Section 2, our criterion is especially useful when applied for targeted checking. For the sake of the experiment, we exclude from our analysis queries issued within declared rendering sections of TOUCHDEVELOP scripts, as they are very frequently executed (every time a page is re-rendered), guaranteed to have no side effects on the program state, and are almost always harmless in practice. We check serializability for all other events.

6.3 Experimental Set-up

We analyzed 33 different applications, which are summarized in Figure 5. 24 of them were written by regular TOUCHDEVELOP users; 6 were written by Microsoft employees to showcase or test the cloud functionality of TOUCHDEVELOP (marked with † in Figure 5). In addition, we analyzed 3 scripts where we fixed some of the bugs that we found (marked with ‡ in the table).

Each application is exercised on two client nodes in parallel via our own random exploration tool for roughly 3 minutes. For 4 games (marked with $*$ in Figure 5) more involved interaction was required, and we executed some of the operations manually. The clients are independently restarted at random during the execution to achieve a realistic overlap in their lifetimes. Both clients are located in Europe while communicating through a data center in the US.

6.4 Analysis Results

We compare our serializability criterion to commutativity races, the most closely related criterion applicable in this setting. A commutativity race [13] is a pair of non-commutative actions unrelated by causality. Their absence is a sufficient condition for serializability under causal consistency with atomic visibility. To see this, observe that (a) under atomic visibility, every cycle in the DSG contains at least one $\ominus$ edge, since $\mathsf{ar} \cup \mathsf{po} \cup \mathsf{vi}$ is acyclic by definition (the system guarantees causal arbitration [7]), (b) $\oplus$ edges cannot introduce cycles as mutual dependency of transactions is impossible under atomic visibility, and (c) every $\ominus$ edge forms a commutativity race under causal consistency.

Figure 5 shows the result of our experiment. Columns **Events** and **Trans.** denote the number of events and transactions, resp., executed within the analyzed schedule. Column **Time [s]** contains the time it took to analyze the schedule on a system equipped with an Intel Core i7-4600U CPU with 2.10GHz and 12GB of memory.

We define the number of serializability violations in a program as the number of $\ominus$ edges involved in cycles in the DSG, mapped from events down to program locations. This is a natural metric, as it overapproximates the number of operations whose order must be fixed by synchronization to resolve the violation. Furthermore, it makes the number of commutativity races (column CR in the table) and serializability violations (column SV) comparable, since each serializability violation in this sense is also a commutativity race.

6.5 Discussion

The experiments show that significantly fewer serializability violations than commutativity races are reported for 21 of the 33 applications; the remaining 12 exhibit neither commutativity races nor serializability violations. Overall, we detect 75% fewer serializability violations than commutativity races. In particular, 21 applications contain commutativity races, but only 8 contain serializability violations. This means that the programmer has to inspect significantly fewer program locations when evaluating the serializability of a system, often none at all. In most cases where commutativity races are reported but no serializability violations, conflicting updates remain unobserved, which is correctly detected by the use of cycle detection and absorption.

We did not find a false alarm among the serializability violations, in the sense that every serializability violation actually caused the replication system to return inconsistent values to the application. In four applications, these data inconsistencies had no effect on the overall application functionality and can, therefore, be considered harmless. In the other four applications that contained serializability violations, the analysis revealed bugs that are likely to be fixed by the developers. In the following, we discuss two of them. We propose bug-fixes and show that establishing their correctness requires precise serializability checking.

Tetris One bug appears in the game "tetris", in which the following program fragment is executed when a new high score is to be saved to the replicated store:

```
if (curScore > cloud.highScore)
  cloud.highScore := curScore
```

Here, a high score of the player's account is stored in a cloud integer. When a game is completed, its score is compared to the local replica of `highScore`. If it is larger, `highScore` is overwritten. The update is later propagated to other clients, potentially overwriting *higher* scores achieved on other clients. When the above transaction is executed concurrently by two clients, the execution schedule shows both a commutativity race and a serialization violation and is thereby detected by ECRACER (see Figure 5, id "gcane").

Implementing a fix is not trivial, as there is no atomic max-function on cloud integers in TOUCHDEVELOP. A fix can instead make use of high-level data structures to store all scores instead of only the first:

```
var scoreRec := cloud.scores.add_row
scoreRec.val := curScore
while (!scoreRev.val.confirmed) sleep(0.2)
var highScore = 0
foreach (s in cloud.scores)
  if (s.val > highScore) highScore = s.val
  else s.delete
```

The fix adds the newest score to a replicated list and then waits until the update is appended to the global prefix. Finally, it selects the highest value among all values stored in the list and deletes all

ID	Name	Category	Events	Trans.	CR	SV	$\frac{\text{SV}}{\text{CR}}$ [%]	Time [s]
sxjua	Cloud Paper Scissors †	Game	244	96	7	2	29	0.066
uvlma	Color Line ∗	Game	5	4	1	0	0	0.003
ycxbc	guess multi-player demo †	Game	293	66	3	1	33	0.104
kqfnc	HackER	Game	115	91	12	6	50	0.181
ohgxa	keyboard hero ∗	Game	2	2	0	0	-	0.001
wccqepeb	Online Tic Tac Toe Multiplayer	Game	565	184	57	17	30	0.324
uvjba	pentix ∗	Game	6	6	3	0	0	0.002
padg	sky locale ∗	Game	347	266	2	1	50	0.118
–	sky locale ∗ ‡	Game	264	195	0	0	-	0.047
gcane	tetris ∗	Game	8	4	2	1	50	0.002
–	tetris ∗ ‡	Game	14	8	1	0	0	0.003
fqaba	Chatter box	Social	131	75	3	0	0	0.020
etww	Contest Voting †	Social	57	57	0	0	-	0.065
eijba	ec2 demo chat †	Social	72	36	0	0	-	0.009
gbtxe	Hubstar	Social	263	183	0	0	-	0.120
nggfa	instant poll †	Social	81	81	0	0	-	0.019
qnpge	metaverse	Social	20	4	3	0	0	0.005
ruef	Relatd	Social	118	65	7	0	0	0.014
cvuz	Super Chat	Social	170	58	0	0	-	0.029
wbuei	unique poll	Social	166	143	2	0	0	0.071
qzju	cloud card	Tool	32	8	0	0	-	0.006
kzwue	Cloud Example	Tool	178	170	2	1	50	0.099
blqz	cloud list †	Tool	302	261	2	0	0	0.082
qwidc	Events	Tool	1458	80	5	2	40	0.772
–	Events ‡	Tool	520	65	0	0	-	0.158
nvoha	expense recorder †	Tool	67	60	3	0	0	0.007
wkvhc	Expense Splitter	Tool	25	14	0	0	-	0.007
kmac	FieldGPS	Tool	12	12	1	0	0	0.005
kjxzcgcv	NuvolaList 2	Tool	297	223	6	0	0	0.340
eddm	Save Passwords	Tool	345	259	0	0	-	0.118
cavke	TouchDatabase	Tool	232	58	0	0	-	0.048
qzeua	TouchDevelop Jr.	Tool	64	49	1	0	0	0.029
whpgc	Vulcanization calculator	Tool	54	30	1	0	0	0.009

Figure 5: Result of the dynamic analysis of 33 TOUCHDEVELOP applications

others. The synchronization in line 4 is required to not incorrectly determine that the new score is a high score, while some other client submitted a better high score, arbitrated before ours. The fix still exhibits a commutativity race between the inserts to the list. However, there is no serialization violation, as the program is serializable.

Sky Locale The "Sky Locale" quiz game allows a user to overwrite the account of an existing user, because of an incorrect uniqueness check, which is detected by our analysis. The essential problem is embodied by the following code:

```
1  if (!Users.at(name).Created) {
2    Users.at(name).Created := true
3    // ...
4  }
```

The code tries to enforce a uniqueness constraint over user names. Here, it is possible that two clients reserve the same name after concurrently reading `false` in line 1. A fix can be derived by reserving the name, forcing synchronization with the other clients, and checking if we have won the race for the name reservation:

```
1  Users.at(name).Created := client_id
2  while (!Users.at(name).Created.confirmed)
3    sleep(0.2)
4  if (!Users.at(name).Created == client_id)
5    // ...
```

Our analysis reports correctly that the fixed version does not contain a serializability violation. In contrast, it does contain a commutativity race between two instances of the update in line 1. However, this race was not triggered during the run of the dynamic analysis and is, thus, not reported in Figure 5.

7. Application: Developing Clients of Weakly Consistent Databases

Developing clients of eventually consistent data stores is a challenging problem. Adding too much synchronization to the program will generally reduce performance, while insufficient synchronization can lead to serializability violations and, thus, unintended application behaviors. In this section, we show that our dynamic analysis can guide the developer towards a correct and efficient implementation. Starting with little or no synchronization, developers can use our analysis to detect serializability violations and then fix these violations by adding more synchronization until the required consistency constraints are met.

To illustrate this application of our analysis, we implemented a common database benchmark on top of an eventually consistent data store and used the analysis to determine the necessary synchronization. In our experiment, our analysis always reported violations that lead to real synchronization problems. Moreover, the analysis correctly classifies all of our fixes as serializable. The resulting implementation is significantly faster and more scalable than a solution with naive synchronization.

In our case study we use RIAK [19], a distributed key-value data store based on a design similar to Amazon's Dynamo [12]. Riak replicates data across a cluster of nodes and keeps it eventually consistent. Operations are typically performed in a highly available manner, where queries contact only a subset of the replicas, and updates return to the client before being confirmed by all nodes. To resolve update-update conflicts in a convergent manner, RIAK provides implementations of several conflict-free replicated data types [30] such as counters, sets, flags, maps, and last-writer-wins registers.

7.1 Dynamic Analysis of Riak-Backed Applications

We integrate our runtime instrumentation as a shim layer around the official Python client library of Riak. This layer serves two purposes: (a) if the dynamic analysis is enabled, the layer records all executed operations of the client application to an independent database, and (b) it gives the developer the ability to provide lightweight specifications in addition to the purely operational API of Riak.

Recorded Information As in Section 6, we require the knowledge of visibility vi, arbitration ar, and program order po, as well as being able to check commutativity ♯ and absorption ▷ between events. po can be trivially determined by sequentially numbering all operations performed by the same client and recording it. To check commutativity and absorption, we record the arguments and return values of each event.

In contrast to Section 6, here visibility vi cannot be tracked using vector clocks: Vector clocks can be used only for transitively closed relations, while visibility in Riak is not. Furthermore, updates are only guaranteed to become atomically visible on a per-key basis. That is why we track visibility information for every stored value separately. Each value is embedded in a Riak-DT-Map [6], along with a set of unique identifiers of all the updates applied to the value. These identifiers correspond directly to vi edges in the DSG. Since changes to the same map are made atomically visible, a client has observed an update if and only if that update's identifier is in the set. Deletions are not performed directly, but instead, the value-embedding map also contains a flag that marks the record as deleted.

Using this instrumentation, the data stored in the database grows linearly in the number of operations performed on the database, which is permissible during testing but prohibitive for production use. This restriction can be partially lifted by making further assumptions: For example, by assuming that clients remain connected to the same node, implying that the set of observed updates is monotonically increasing, one can track observed updates for each client separately and prune observed updates from the observed sets. We do not apply such a technique in our evaluation, as short execution traces suffice for our purposes.

Lightweight Specifications If the dynamic analysis is used without any developer annotations, every operation will be observed as a single-operation transaction. In that case, ECRacer essentially checks for sequential consistency [22] of the recorded execution. Our client library provides two ways of expressing the developer's intent: (a) one may designate that a set of operations forms a transaction, that is, are expected to have *serializable* behavior; (b) the developer may *exclude* query operations from the serializability checking. We then allow such operations to return inconsistent values, as described in Section 2.

Offline Analysis The offline analysis is performed in the same manner and, in fact, with the same core implementation as in Section 6, despite that it targets different systems. ECRacer is extended with commutativity and absorption specifications for all operations provided by Riak. Here, we use the semantics of Riak's CRDTs to derive the arbitration order. For example, for an increment-only counter, all updates are unordered as they all commute; therefore the arbitration order is empty. For an add-wins set, concurrent adds are unordered, concurrent removes are unordered, and every add is ordered after all concurrent removes. Finally, for a last-writer-wins register, all updates to the register are totally ordered by their physical timestamps.

7.2 Analyzing TPC-C using ECRacer

TPC-C [36] is one of the most well-known database benchmarks. It defines a database-backed whole-sale supplier application, featuring among others payment, delivery, order status, stock level status, and order creation transactions. It is typically implemented by vendors of databases that provide serializable transactions but has also been extensively used for the benchmarking of weakly synchronized distributed databases [2].

Version	#Txns	#Events	Time [s]	#Viol.
1	280	7197	28.064	9
2	307	6907	24.806	6
3	365	7236	20.938	2
4	441	6903	7.678	1
5	475	7192	8.113	1
6	449	6903	6.823	0

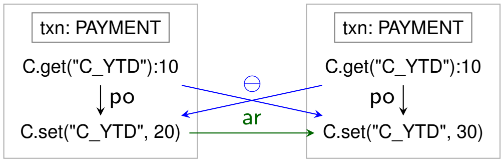
Figure 6: Analysis results for each version of TPC-C. **#Txns** is the number of transactions, **#Events** is the number of events recorded, **Time** is the total analysis time in seconds, and **#Viol.** is the number of detected violations

We use our analysis to derive a correctly synchronized version of a TPC-C implementation, by iteratively eliminating violations detected by our analysis. Initially, we start with an implementation of TPC-C (in Python), loosely based on the sample programs given in Appendix A of the TPC-C specification [36]. These programs use a standard table-based data model and assume support for serializable transactions from the database.

We deliberately do *not* partition the data horizontally by the warehouse, as is common [2]. Our goal is to derive an implementation that can scale up a single warehouse and even a single district to a large number of servers.

Each version of the implementation is run with the previously described runtime instrumentation for 20 seconds with 3 clients in parallel on a minimal three node setup of Riak on a remote server. The number of transactions and operations executed, the analysis time and the detected violations are shown in Figure 6. The analysis was run on a system equipped with an Intel Core i7-4600U CPU with 2.10GHz and 12GB of memory.

Version 1 In the first version, the analysis detects 9 violations. Three of those are due to increments being performed in a non-atomic way:

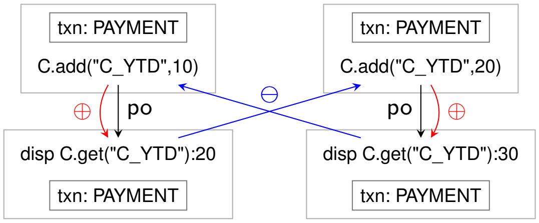


In this and the coming figures, the boxes represent nodes (that is, transactions) in the dependency serialization graph. Insides the boxes, we show the fragment of the events in the transaction that are central to the described serializability violation. For illustration, we show relations as arrows between events; however, in the DSG, these are edges between nodes. Consequently, the figure above shows a cycle in the DSG between the two nodes representing the two instantiations of the PAYMENT transaction and, thus, a serializability violation.

Observing the violation above, a developer can easily see that the left update to the customer's sum of all year-to-date payments is lost, as it is overwritten by the right update. They can then check whether serializability is required for this set of operations. In this particular case, we refer to the Consistency Requirements section of the TPC-C benchmark [36] to see that losing an update to C_YTD may violate Consistency Requirement 12 of the database and therefore the serialization violation above points to an actual synchronization problem.

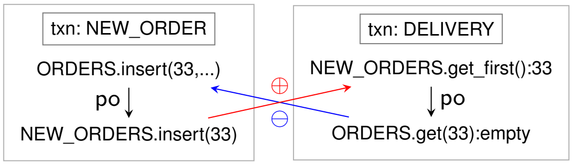
The problem can easily be solved without coordination, by replacing C_YTD by a CRDT counter with commutative increments.

Version 2 The result of replacing non-atomic increments of counters by commutative counter increments, leads to the following execution fragment:



Note the difference to the previous fragment: Here, the two increments are unordered by arbitration (as they commute), and they do not absorb each other. Therefore, we get two ⊖ edges, forming a cycle in the DSG. In any serialization, one of the get queries must read the sum 40, assuming that the initial value was 10. The two queries are used only to display the year's sum of payments on the terminal; the requirements document [36] does not require those displayed values to be consistent. Therefore, we chose performance over strong consistency in this case by adding a lightweight annotation to exclude the queries in the above figure from the serializability checking. With similar reasoning, we can resolve several other violations.

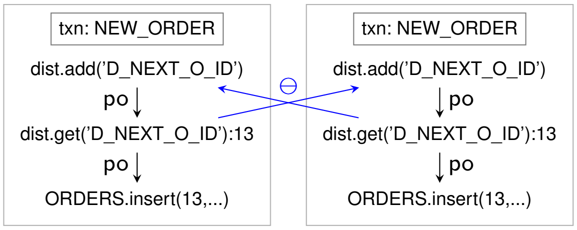
Version 3 After excluding the violations above through targeted checking, ECRACER still detects two violations. One of those is due to partial observation of transactions, as in the following cycle between NEW_ORDER and DELIVERY:



The left transaction inserts a new order into the ORDERS table and a foreign key to that order into the NEW_ORDERS table. The right transaction observes the foreign key but does not observe the corresponding order record. The DSG defines that, in a serialization, the order insertion must follow the order retrieval to make its return value legal, but also requires the foreign key insertion to be ordered before the foreign key retrieval, creating a cycle with the program order.

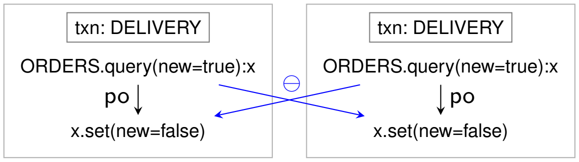
To solve the problem, we need to change our application such that the modifications to ORDER and NEW_ORDER are reflected atomically at each replica. RIAK and most other stores provide atomicity for all updates applied to the same row. This means, we can solve the problem by denormalizing the data and combining tables into a common CRDT data structure, which is stored in a single row of the database. In this example, we represent the ORDERS record by a RIAK-DT-Map [6], and embed the NEW_ORDER flag into the entries of that map. Similarly, the lines of the order are embedded as a set CRDT in the ORDERS map.

Version 4 In Version 4 of the implementation, only one violation is detected: Two parallel increments to D_NEXT_O_ID (the district's next, serially assigned order number) and two queries to that value in the same NEW_ORDER transaction:



Clearly, this behavior is non-serializable, as it should not be possible for both transaction instances to read value 13 in the second operation. This problem is classic for TPC-C and was previously shown to be impossible to implement without coordination [2]. To resolve the problem (which is not directly possible in RIAK), we use atomic counters, externally synchronized by ZooKeeper [17], a high-performance service for distributed synchronization.

Version 5 After running the Version 4 several times, the analysis reported potential double delivery, a rare circumstance due to the infrequent execution of the delivery transaction:



Here, the transaction receives all new orders from the database and subsequently disables the new flag. While the implementation would behave correctly for serial schedules, in RIAK it may lead to double deliveries. We solve the problem by exploiting the rarity of the delivery transactions: We can force its execution on a single server and lock it locally, without compromising performance.

Version 6 In the final version, no serializability violation is detected. While ECRACER, being a dynamic analysis, cannot provide a guarantee about the absence of violations, one can gain significant confidence by creating bad-case scenarios (network partitions, node failure, etc.) during the dynamic analysis.

7.3 Scalability

Finally, we evaluate the throughput of our incrementally derived, serializable implementation to an implementation with straightforward synchronization. The former, labeled *Custom* in Figure 7 corresponds to Version 6 from the previous subsection, while the latter, labeled *Locked*, corresponds to Version 1, extended with a somewhat naive synchronization, where clients lock the parts of the database they access using ZooKeeper primitives.

We run both versions on 4 to 10 `m4-large` Amazon EC2 instances with 2 virtual cores and 8GiB of RAM each, running clustered instances of both RIAK and ZooKeeper. RIAK is run in its default configuration with triple replication and Apache SOLR providing advanced querying on top of the key-value store.

Our benchmark follows the standard usage of distributed databases: it replicates data across nodes for failure tolerance, and it does not use stored procedures to implement transactions. It is therefore not directly comparable to optimized implementations, and much higher throughputs can be achieved with further domain knowledge. However, the benchmark clearly shows that the manual synchronization derived from our analysis result, without any domain knowledge, scales much better than the naive locking approach.

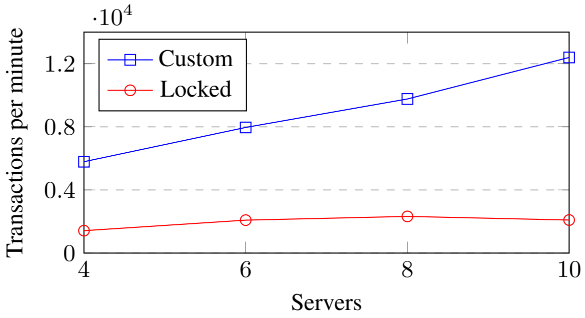


Figure 7: Performance comparison of Version 6 (*Custom*) and a primitively synchronized variant of Version 1 (*Locked*)

8. Related Work

Conflict serializability is a restriction of serializability that can be checked in polynomial time [28]. It is based on reasoning about conflicts between basic reads and writes, but can directly be lifted to commutativity (see, e.g., [38]). However, conflict serializability assumes sequentially consistent [22] histories and is therefore not directly applicable to weakly consistent systems.

Several works define serializability conditions on executions on data stores with various weak guarantees, for example, snapshot isolation [14], causal consistency [3], as well as a variety of weak memory models (e.g., [5, 27, 32]). As with our criterion, these are typically based on detecting cycles in graphs involving some notion of dependency and anti-dependency. The main differences to our work are that (a) they assume stronger consistency guarantees from the data store, and (b) they use low-level read and write reasoning instead of algebraic reasoning. Our work is a generalization of these previous criteria, which makes them applicable to a broader class of real-world systems.

Zellag and Kemme [39] use a criterion similar to Fekete et al. [14] to quantify the anomalies in applications running against eventually consistent data stores. However, they do not prove the criterion correct w.r.t. eventual consistency and again reason only about reads and writes.

Commutativity races [13] are sufficient for serializability if the system provides causal consistency. However, they are less precise than our criterion (see our comparison in Section 6) and are not sufficient for serializability under the weaker eventual consistency.

Several works suggest reasoning about the preservation of integrity invariants in weakly consistent data stores (see, e.g., [2]). While invariant-based reasoning can permit more behaviors than serializability and can also lead to additional performance gains, it demands detailed specifications, which are notoriously difficult to obtain. Our work instead uses a generic correctness condition.

9. Conclusion and Future Work

We presented a new serializability criterion for eventually consistent data stores and demonstrated its usefulness via two dynamic analyses. The criterion generalizes the classic notion of conflict serializability to high-level data types by leveraging the concepts of commutativity and absorption. Our evaluation suggests that the concepts and systems presented in this work are useful for building correct and efficient applications on top of eventually consistent data stores.

We expect this work to be a starting point for further research into both the correctness of clients of eventually consistent data stores as well as other concurrent programs using high-level data types. Our criterion is also well-suited to be extended to static analyses using suitable abstraction, which we plan to develop as future work.

In the reads and writes setting, previous work [3] has found that only certain restricted classes of cycles can occur in dependency serialization graphs under guarantees stronger than eventual consistency. It would be interesting to discover if one can find similar restrictions for arbitrary operations.

References

[1] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 67–78, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0506-6. URL http://dl.acm.org/citation.cfm?id=846219.847380.

[2] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014. ISSN 2150-8097. doi:10.14778/2735508.2735509. URL http://dx.doi.org/10.14778/2735508.2735509.

[3] G. Bernardi and A. Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR'16: International Conference on Concurrency Theory*, 2016.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[5] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pages 533–553, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-37035-9. doi:10.1007/978-3-642-37036-6_29. URL http://dx.doi.org/10.1007/978-3-642-37036-6_29.

[6] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak DT map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2716-9. doi:10.1145/2596631.2596633. URL http://doi.acm.org/10.1145/2596631.2596633.

[7] S. Burckhardt. *Principles of Eventual Consistency*, volume 1 of *Foundations and Trends in Programming Languages*. now publishers, October 2014. URL http://research.microsoft.com/apps/pubs/default.aspx?id=230852.

[8] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi:10.1145/2535838.2535848.

[9] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming*, volume 37 of *ECOOP 2015*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.568. URL http://drops.dagstuhl.de/opus/volltexte/2015/5238.

[10] A. Cerone, G. Bernardi, and A. Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In L. Aceto and D. de Frutos Escrig, editors, *26th International Conference on Concurrency Theory*, volume 42 of *CONCUR 2015*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-91-0. doi:http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.58. URL http://drops.dagstuhl.de/opus/volltexte/2015/5375.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008. ISSN 2150-8097. doi:10.14778/1454159.1454167. URL http://dx.doi.org/10.14778/1454159.1454167.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi:10.1145/1294261.1294281.

[13] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 305–315, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi:10.1145/2594291.2594322.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2): 492–528, June 2005. ISSN 0362-5915. doi:10.1145/1071610.1071615. URL http://doi.acm.org/10.1145/1071610.1071615.

[15] G. Gierz, K. Hofmann, K. Keimel, J. Lawson, M. Mislove, and D. Scott. *Continuous Lattices and Domains*. Number 93 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003. ISBN 9780521803380. doi:10.1017/CBO9780511542725.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2):51–59, June 2002. ISSN 0163-5700. doi:10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[18] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. RFC 677, RFC Editor, january 1975. URL http://www.rfc-editor.org/rfc/rfc677.txt.

[19] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0516-7. doi:10.1145/1900160.1900176. URL http://doi.acm.org/10.1145/1900160.1900176.

[20] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010. ISSN 0163-5980. doi:10.1145/1773912.1773922. URL http://doi.acm.org/10.1145/1773912.1773922.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi:10.1145/359545.359563.

[22] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. ISSN 0018-9340. doi:10.1109/TC.1979.1675439. URL http://dx.doi.org/10.1109/TC.1979.1675439.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi:10.1145/2043556.2043593. URL http://doi.acm.org/10.1145/2043556.2043593.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482657.

[25] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[26] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 278–324. Springer Berlin Heidelberg, 1987. ISBN 978-3-540-17906-1. doi:10.1007/3-540-17906-2_30.

[27] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5. doi:10.1007/978-3-642-14107-2_23.

[28] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979. ISSN 0004-5411. doi:10.1145/322154.322158. URL http://doi.acm.org/10.1145/322154.322158.

[29] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro. SwiftCloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, SRDSW '14, pages 30–33, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-7361-3. doi:10.1109/SRDSW.2014.33. URL http://dx.doi.org/10.1109/SRDSW.2014.33.

[30] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011. URL https://hal.inria.fr/inria-00555588.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. URL http://dl.acm.org/citation.cfm?id=2050613.2050642.

[32] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2): 282–312, Apr. 1988. ISSN 0164-0925. doi:10.1145/42190.42277.

[33] S. Sivasubramanian. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi:10.1145/2213836.2213945. URL http://doi.acm.org/10.1145/2213836.2213945.

[34] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi:10.1145/2043556.2043592. URL http://doi.acm.org/10.1145/2043556.2043592.

[35] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 49–60, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0941-7. doi:10.1145/2048237.2048245. URL http://doi.acm.org/10.1145/2048237.2048245.

[36] Transaction Processing Performance Council. TPC-C benchmark, revision 5.11, Feb. 2010.

[37] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi:10.1145/1111037.1111067. URL http://doi.acm.org/10.1145/1111037.1111067.

[38] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, Dec. 1988. ISSN 0018-9340. doi:10.1109/12.9728.

[39] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1761-0. doi:10.1145/2391229.2391235. URL http://doi.acm.org/10.1145/2391229.2391235.