Université de Paris

École Doctorale de Sciences Mathématiques de Paris Centre (386)
Institut de Recherche en Informatique Fondamentale (UMR 8243)

Automated Formal Testing of Storage Systems and Applications

par Ranadeep BISWAS

Thèse de doctorat d'informatique

Dirigée par Ahmed BOUAJJANI
Et par Constantin ENEA

Présentée et soutenue publiquement *le 30 Mars 2021*
Devant un jury composé de :

Bengt JONSSON	*Professor*	Uppsala University, Suède	Rapporteur
Ilya SERGEY	*Associate Professor*	Yale-NUS College, Singapore	Rapporteur
Mihaela SIGHIREANU	*Professeure*	ENS Paris-Saclay	Examinatrice
Noam RINETZKY	*Associate Professor*	Tel Aviv University, Israel	Examinateur
Viktor VAFEIADIS	*Tenured Researcher*	Max Planck Institute for Software Systems, Kaiserslautern	Examinateur
Ahmed BOUAJJANI	*Professeur*	Université de Paris	Directeur
Constantin ENEA	*Maitre de conférences (HDR)*	Université de Paris	Co-directeur

Abstract

As *internet* grows to be cheaper and faster, distributed software systems and applications are becoming more and more ubiquitous. Today they are the backbone of a large number of online services like banking, e-commerce, social networking, etc. As the popularity of these softwares increases, it is very important that they ensure strong levels of reliability and security.

Modern distributed software is centered around using large-scale storage systems for storing and retrieving data. To ensure persistence and availability of data in the presence of failures, these systems maintain data in multiple copies that are stored on different nodes in the network. Then, for performance reasons, these copies are allowed to (temporarily) diverge, an instance of the so-called *weak-consistency*, which makes the semantics of concurrent accesses to data quite complex.

Over the recent years, many solutions for implementing *weakly-consistent* storage systems have been proposed. These implementations are most often very complex and error-prone. The specific levels of weak consistency they ensure are most often described only informally, which makes it difficult to reason about them. Moreover, in many cases, there are significant discrepancies between the guarantees claimed in their documentation and the guarantees that they really provide.

The objective of this dissertation is to propose algorithmic techniques for *automated testing* of weakly-consistent distributed systems against *formal specifications*. We focus on an important class of distributed data types, called *Conflict-Free Replicated Data Types* (*CRDT*s for short), that include many variations like registers, flags, sets, arrays, etc., and on *Transactional Systems (Databases)*, which enable computations on shared data that are isolated from other concurrent computations and resilient to failures. We introduce formal specifications for such systems and investigate the asymptotic complexity of checking whether a given execution conforms to such specifications. We also study the problem of testing applications that run on top of weakly-consistent transactional systems, introducing an mock in-memory storage system that simulates the behaviors of such systems according to their formal specifications.

Keywords: Formal Methods, Concurrency, Distributed Systems, Databases, Automated Testing, Weak Consistency, Replicated Data Types, Transactions, Isolation Levels, Complexity

Résumé

À mesure que *l'internet* devient moins cher et plus rapide, les systèmes et les applications logicielles distribués deviennent de plus en plus omniprésents. Aujourd'hui, ils sont à la base d'un très grand nombre de services en ligne tels que les banques, le commerce électronique, les réseaux sociaux, etc. Au fur et à mesure que la popularité de ces logiciels augmente, il est très important qu'ils garantissent des niveaux élevés de fiabilité et de sécurité.

Les logiciels distribués modernes sont centrés sur l'utilisation de systèmes de stockage à grande échelle pour stocker et manipuler des données. Pour assurer la persistance et la disponibilité des données en présence de pannes, ces systèmes maintiennent les données en plusieurs copies stockées sur différents nœuds du réseau. Pour des raisons de performances, ces copies peuvent diverger (temporairement), une instance de la soi-disant *cohérence faible*, ce qui rend la sémantique des accès concurrents aux données très complexe.

Au cours des dernières années, de nombreuses solutions pour implémenter des systèmes de stockage à *cohérence faible* ont été proposées. Ces implémentations sont le plus souvent très complexes et sujettes aux erreurs. Les niveaux spécifiques de cohérence faible qu'ils assurent ne sont le plus souvent décrits que de manière informelle, ce qui rend difficile le raisonnement sur leurs correction. De plus, dans de nombreux cas, il existe des écarts importants entre les garanties mentionnées dans leur documentation et les garanties qu'elles fournissent réellement.

L'objectif de cette thèse est de proposer des techniques algorithmiques pour le *teste automatisé* de systèmes distribués à cohérence faible par rapport à des *spécifications formelles*. Nous étudions une classe importante de types de données distribués, appelés *types de données répliqués sans conflit* (*CRDT*), qui inclut de nombreuses variantes comme des registres, des ensembles, des tableaux, etc., et des *systèmes (bases de données) transactionnels*, qui permettent des calculs sur des données isolés des autres calculs concurrents et tolérants aux pannes. Nous introduisons des spécifications formelles pour de tels systèmes et nous étudions la complexité asymptotique de la vérification de la correction d'une exécution donnée par rapport à ces spécifications. Nous étudions également le problème du teste des applications qui s'exécutent sur des systèmes transactionnels à cohérence faible, en introduisant un système de stockage en mémoire qui simule les comportements de ces systèmes par rapport à leurs spécifications formelles.

Mots-clés: Méthodes formelles, Concurrence, Systèmes distribués, Bases de données, Teste automatisé, Cohérence faible, Types de données répliqués, Transactions, Complexité.

Acknowledgments

I would like to express my deepest appreciation to my advisors, teachers, colleagues, family, and friends, without whom this dissertation would not have been possible - who accompanied and helped me through my Ph.D. journey.

My deepest gratitude to my supervisors Dr. Ahmed Bouajjani and Dr. Constantin Enea - without their assistance, advice, and motivation through the past three and half years, this thesis would not have been possible, especially in the time of this pandemic. They taught me the nitty-gritties of verification of distributed systems, and most importantly how to challenge myself to become a good researcher which I will treasure forever. I could not have imagined having better advisors and mentors for my doctoral study.

I would also like to thank the jury: Dr. Bengt Jonsson, Dr. Ilya Sergey, Dr. Mihaela Sighireanu, Dr. Noam Rinetzky, Dr. Viktor Vafeiadis. Bengt and Ilya were very kind to review my dissertation in this challenging time - Thank you.

A good support system is important. Thanks to my colleagues and friends Sidi Mohamed Beillahi, Berk Cirisci, Suha Orhun Mutluergil, Rachid Zennou, for being there every step of the way and for making IRIF so friendly. Whenever I was stuck with some problems, our discussions helped me progress. I would also like to thank the administrative staff at ED 386, IRIF, and UFR Informatique for their help during my stay for the research.

My Ph.D. would not have happened without the amazing professors of Chennai Mathematical Institute (Chennai, India). Thank you Dr. Balaguru Srivathsan, for your guidance, advice, and lectures that inspired me to choose Computer Science as my career.

Most importantly, my family - without whom, this would never have been possible. My parents Kanyakumarika and Ramtanu Biswas, grandparents Namita and Krishnapada Paik, my brothers Rudradeep and Suvadeep, and uncle Arup Paik - your love, sacrifice, and unconditional support helped me reach where I am right now.

Last but not the least, I would like to thank my friends who made this journey interesting and fun. Poulomi, Subhajit, Mouli, Sanchari, Kanchan, Sougata, Abhishek, Arpita - thank you for everything. Darshana - thank you for being my first friend in Paris, so far away from home. Anjum - thank you for adopting me as a brother, you are the sister I never had. Melisa and Simone - thank you for keeping me sane with all the best experiences in Europe, you are the best of friends that I could have ever asked for. Thank you Paula for all the love and support.

In the end, I would like to thank each and everyone in my life, who made me a better person and helped me reach this position.

Thank you!

Contents

1 Introduction

As *internet* grows to be cheaper and faster, distributed software systems and applications are becoming more and more ubiquitous. Today they are the backbone of a large number of online services like banking, e-commerce, social networking, etc. As the popularity of these softwares increases, it is very important that they ensure strong levels of reliability and security.

Distributed software is deployed over multiple nodes connected through a network, e.g., the *internet*. Data is typically *replicated* at multiple nodes, in order to guarantee availability and fast response to user requests. A user connects to the node *nearest* to them and that node serves its requests. This way the system reduces response time and distributes the workload to multiple nodes. Also, if some node goes offline, the system remains available since other nodes can still serve users.

While data replication is a solution to improving availability and scalability, it actually offers a trade off. As we allow concurrent modifications of data at multiple nodes, they still need to synchronize among them and maintain a meaningful or *consistent* view of data. A pessimistic approach to maintaining consistency, based on global locks (or other synchronization protocols like 2-Phase-Commit), defeats the whole purpose of replication, because taking a lock over a multiple distributed nodes means more communication and slow response time.

Over the recent years, many solutions for implementing *weakly-consistent* distributed systems have been proposed. Such systems allow different nodes to store different versions of data in favor of scalability, thereby violating notions of strong consistency (all nodes store the same data at all times) that could be maintained using the pessimistic approaches mentioned above. The specific levels of consistency these systems ensure are most often described only informally, which makes it difficult to reason about them. Moreover, in many cases, there are significant discrepancies between the guarantees claimed in their documentation and the guarantees that they really provide.

The objective of this dissertation is to propose algorithmic techniques for *automated testing* of weakly-consistent distributed systems against *formal specifications*. We focus on an important class of distributed data types, called *Conflict-Free Replicated Data Types* (*CRDT*s for short), that include many variations like registers, flags, sets, arrays, etc., and on *Transactional Systems (Databases)*, which enable computations on shared data that are isolated from other concurrent computations and resilient to failures. We introduce formal specifications for such systems and investigate the asymptotic complexity of checking whether a given execution conforms to such specifications. We also study the problem of testing applications that run on top of weakly-consistent transactional systems, introducing an mock in-memory storage system that simulates the behaviors of such systems according to their formal specifications.

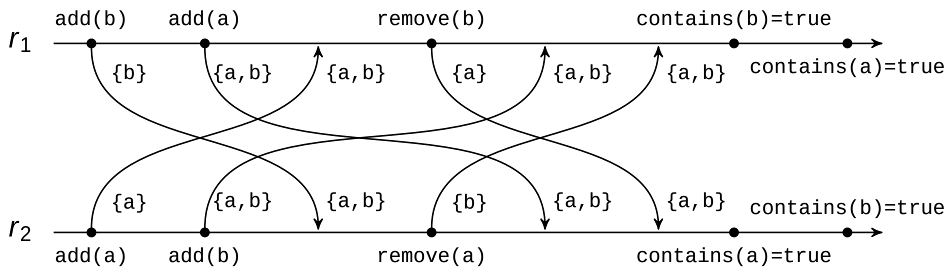


Figure 1.1: A non-linearizable OR-Set execution. Edges represent propagation of updates. Each replica is annotated with labels showing the evolution of the set object after each update.

1.1 Conflict-Free Replicated Data Types

Conflict-Free Replicated Data Types (CRDTs) [87] represent a methodological approach to the problem of retaining some form of data-Consistency and Availability under network Partitions (CAP), famously known to be an impossible combination of requirements by the CAP theorem of Gilbert and Lynch [55]. CRDTs are data types designed to favor availability over consistency by replicating the type instances across multiple nodes of a network, and allowing different nodes to temporarily have different views of the same instance. However, CRDTs guarantee that the different states of the multiple nodes will *eventually* converge to a unique state common to all nodes [26, 87]. Importantly, this *convergence property* is intrinsic to the data type design and in general no synchronization is needed among nodes, hence achieving availability.

A client, i.e. a program issuing calls to a data type instance, connects to any node holding a copy of the instance, called *replica*, and performs the operation in that replica. The state of the instance is read only at that replica, and if the state needs to be changed as part of the operation, an update is generated, which will be *asynchronously* propagated to all the other replicas. When updates eventually reach all replicas, they may be received in different orders by different replicas. To ensure convergence, conflicts between concurrent updates need to be resolved. This is quite non-trivial and an important source of complexity.

For instance, Figure 1.1 pictures an execution of a CRDT called *OR-Set* [87], a set data type with standard `add(_)`, `remove(_)`, `contains(_)` operations. `add(_)` and `remove(_)` are the only update operations. Two updates are in conflict if they are trying to insert or remove the same element, and possible conflicts are resolved by assuming that an `add(_)` operation will always “win” among multiple conflicting concurrent updates, i.e., it will overwrite their effect. In Figure 1.1, each replica executes the first two `add` operations in isolation (without being aware of operations on the other replica), receives the first `add` update from the other replica, and executes a `remove` operation before receiving the second `add` update from the other replica (as mentioned above, updates are propagated to other replicas asynchronously). The element b, resp., a, is again a member of the set on the top replica, resp., bottom replica, after receiving the last `add` update because the latter is concurrent (not causally related) to the `remove` on the receiving replica and the conflict is resolved by assuming that the `add` wins. This is witnessed by the last two `contains` operations on each replica that both return `true`. Note that this execution is an instance of *weak consistency* since the return values of the `contains` operations cannot be explained using an interleaving of

Data Types	Complexity
Add-Wins Set, Remove-Wins Set	NP-complete
Enable-Wins Flag, Disable-Wins Flag	NP-complete
Last-Writer-Wins Register (LWW)	NP-complete
Multi-Value Register (MVR)	NP-complete
Registers – with unique values	PTIME
Replicated Counters	NP-complete
Replicated Growable Array (RGA)	PTIME

Figure 1.2: The complexity of consistency checking for various replicated data types.

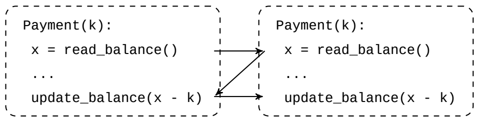


Figure 1.3: A concurrent program. Edges show a non-transactional execution where both reads execute before a write.

these operations (consistent with the order between operations on the same replica) as in classic variations of strong consistency, e.g., sequential consistency [67] or linearizability[60].

In this thesis we study the tractability of checking whether an execution of a CRDT conforms to the intended specification for different classes of data types; Figure 1.2 summarizes some of our results. This problem is particularly relevant as distributed-system testing tools like Jepsen [65] are appearing; without efficient, general consistency-checking algorithms, such tools could be limited to specialized classes of errors like node crashes.

Our study proceeds in two parts. First, to precisely characterize the consistency of various CRDTs, and facilitate symbolic reasoning, we develop novel logical characterizations to capture their guarantees. These characterizations integrate the data type semantics into the consistency guarantees, as opposed to existing formalizations, e.g., [26, 29], of eventual consistency [91], causal consistency [68], sequential consistency, or linearizability, where the data type semantics is a parameter of the consistency specification.

Second, we demonstrate the intractability of several CRDTs by reduction from propositional satisfiability (SAT) problems, and we develop tractable consistency-checking algorithms for individual data types and special cases. Previous work has mostly focused on the problem of checking conformance to *strong* notions of consistency, e.g., checking for sequential consistency [15, 27, 59, 84], serializability [36, 45, 46, 57], or linearizability [28, 44, 70, 95].

1.2 Transactional Systems

Transactions simplify concurrent programming by enabling multiple computations on shared data that are isolated from other concurrent computations and resilient to failures. As an illustrating example, consider the Payment procedure in Figure 1.3 to be executed by two different processes. If we allow the internal read and write operations to be interleaved, we can have a scenario where both reads happen before a write. This would allow a user to pay €200 while his balance decreases only by €100. Executing the code of Payment as a transaction can disable such a behavior since each invocation is executed in isolation without interference from the other invocation. Modern databases provide transactions in various forms corresponding to different tradeoffs between consistency and availability. The strongest level of consistency is achieved with *serializable* transactions [80] whose outcome in concurrent executions is the same as if the transactions were executed atomically in some order. Unfortunately, serializability carries a significant penalty on the availability of the system assuming, for instance, that the database is accessed over a network that can suffer from partitions or failures. For this reason, modern databases often provide weaker guarantees about transactions, formalized by weak consistency models, e.g., causal consistency [68] and snapshot isolation [12].

Implementations of large-scale databases providing transactions are difficult to build and test. For instance, distributed (replicated) databases must account for partial failures, where some components or the network can fail and produce incomplete results. Ensuring fault-tolerance relies on intricate protocols that are difficult to design and reason about. The black-box testing framework Jepsen [63] found a remarkably large number of subtle problems in many production distributed databases.

Testing a transactional database raises two issues: (1) deriving a suitable set of testing scenarios, e.g., faults to inject into the system and the set of transactions to be executed, and (2) deriving efficient algorithms for checking whether a given execution satisfies the considered consistency model. The Jepsen framework aims to address the first issue by using randomization, e.g., introducing faults at random and choosing the operations in a transaction randomly. The effectiveness of this approach has been proved formally in recent work [79]. The second issue is, however, largely unexplored. Jepsen checks consistency in a rather ad-hoc way, focusing on specific classes of violations to a given consistency model, e.g., dirty reads (reading values from aborted transactions). This problem is challenging because the consistency specifications are non-trivial and they cannot be checked using, for instance, standard local assertions added to the client's code.

Besides serializability, the complexity of checking correctness of an execution w.r.t. some consistency model is unknown. Checking serializability has been shown to be NP-complete [80], and checking causal consistency in a *non-transactional* context is known to be polynomial time [21]. In this thesis, we try to fill this gap by investigating the complexity of this problem w.r.t. several consistency models and, in the case of NP-completeness, devising algorithms that are polynomial time assuming fixed bounds for certain parameters of the input executions, e.g., the number of sessions.

We consider several consistency models that are the most prevalent in practice. The weakest of them, *Read Committed* (RC) [12], requires that every value read in a transaction is written by a committed transaction. *Read Atomic* (RA) [31] requires that successive reads of the same variable in a transaction return the same value (also known as Repeatable Reads [12]), and that a transaction "sees" the values written by previous transactions in the same session. In general, we assume that transactions are organized in *sessions* [90], an abstraction of the sequence of transactions performed during the execution of an application. *Causal Consistency* (CC) [68] requires that if a transaction t_1 "affects" another transaction t_2, e.g., t_1 is ordered before t_2 in the same session or

t_2 reads a value written by t_1, then these two transactions are observed by any other transaction in this order. *Prefix Consistency* (PC) [30] requires that there exists a total commit order between all the transactions such that each transaction observes a prefix of this sequence. *Snapshot Isolation* (SI) [12] further requires that two different transactions observe different prefixes if they both write to a common variable.

We establish that checking whether an execution satisfies RC, RA, or CC is polynomial time, while the same problem is NP-complete for PC and SI. Moreover, in the case of the NP-complete consistency models (PC, SI, SER), we show that their verification problem becomes polynomial time provided that, roughly speaking, the number of sessions in the input executions is considered to be fixed (i.e., not counted for in the input size). In more detail, we establish that checking SER reduces to a search problem in a space that has polynomial size when the number of sessions is fixed. (This algorithm applies to arbitrary executions, but its complexity would be exponential in the number of sessions in general.) Then, we show that checking PC or SI can be reduced in polynomial time to checking SER using a transformation of executions that, roughly speaking, splits each transaction in two parts: one part containing all the reads, and one part containing all the writes (SI further requires adding some additional variables in order to deal with transactions writing on a common variable). We extend these results even further by relying on an abstraction of executions called *communication graphs* [33]. Roughly speaking, the vertices of a communication graph correspond to sessions, and the edges represent the fact that two sessions access (read or write) the same variable. We show that all these criteria are polynomial-time checkable provided that the *biconnected* components of the communication graph are of fixed size.

These results rely on a novel specification framework for such criteria which is of independent interest. This framework uses logical constraints, called *axioms*, to characterize the set of executions that conform to a particular consistency level. An execution is modeled using a specific set of relations between events/transactions that describe control-flow or data-flow dependencies: a program order po between events in the same transaction, a session order so between transactions in the same session, and a write-read wr (read-from) relation that associates each read event with a transaction that writes the value returned by the read. These relations along with the events (also called, operations) in an execution are called a *history*. A given history is said to satisfy a consistency model if it admits a *total commit order* between its transactions satisfying a specific set of axioms, which intuitively, define lower bounds on the set of transactions t_1 that *must* precede in commit order a transaction t_2 that is read in the execution.

We provide an experimental evaluation of our algorithms on executions of several production databases, that makes it possible to uncover new bugs or contradictions to their documentation. In particular, we show that, although the asymptotic complexity of our algorithms is exponential in general (w.r.t. the number of sessions), the worst-case behavior is not exercised in practice.

1.3 Applications Using Transactional Systems

Data storage is no longer about writing data to a single disk with a single point of access. Modern applications require not just data reliability, but also high-throughput concurrent accesses. Applications concerning supply chains, banking, etc. use traditional relational databases for storing and processing data, whereas applications such as social networking software and e-commerce

```
// Append item to cart                  // Fetch cart and delete item
AddItem(item i, userId) {               DeleteItem(item i, userId) {
  Begin()                                 Begin()
  key = "cart:" + userId                  key = "cart:" + userId
  cart = read(key)                        cart = read(key)
  cart.append(i)                          cart.remove(i)
  write(key, cart)                        write(key, cart)
  Commit()                                Commit()
}                                       }
```

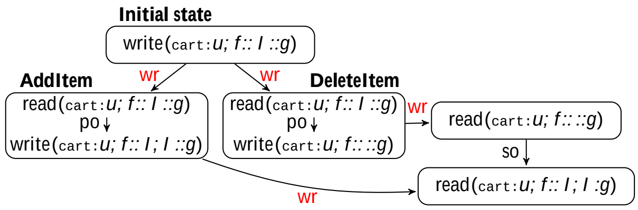


Figure 1.4: A simple shopping cart service.

platforms use cloud-based storage systems (such as Azure CosmosDb [82], Amazon DynamoDb [38], Facebook TAO [23], etc.). We use the term *storage system* to refer to any such database system/service.

Providing high-throughput processing, unfortunately, comes at an unavoidable cost of weakening the guarantees offered to users. Concurrently-connected clients may end up observing different views of the same data. These "anomalies" can be prevented by using a strong *consistency model* such as *serializability*, which essentially offers a single view of the data. However, since serializability requires expensive synchronization and incurs a high performance cost, most storage systems use weaker consistency models, such as RC, CC, or SI. In a recent survey of database administrators [81], 86% of the participants responded that most or all of the transactions in their databases execute at read committed (RC) consistency models.

A weaker consistency model allows for more possible behaviors than stronger consistency models. It is up to the developers then to ensure that their application can tolerate this larger set of behaviors. Unfortunately, weak consistency models are hard to understand or reason about [3, 24] and resulting application bugs can cause loss of business [94]. Consider a simple shopping cart application that stores a per-client shopping cart in a key-value store (*key* is the client ID and *value* is a multi-set of items). Figure 1.4 shows procedures for adding an item to the cart (`AddItem`) and deleting *all* instances of an item from the cart (`DeleteItem`). Each procedure executes in a transaction, represented by the calls to `Begin` and `Commit`. Suppose that initially, a user u has a single instance of item l in their cart. Then the user connects to the application via two different sessions (for instance, via two browser windows), adds l in one session (`AddItem(`l`, `u`)`) and deletes l in the other session (`DeleteItem(`l`, `u`)`). With serializability, the cart can either be left in the state $\{l\}$ (delete happened first, followed by the add) or $\emptyset$ (delete happened second). However, with causal consistency (or read committed), it is possible that with two sequential reads of the shopping cart, the cart is empty in the first read (signaling that the delete has succeeded), but there are *two* instances of l in the second read! The history corresponding to this behavior is given on the bottom of Figure 1.4 (read operations include the read value, and boxes group events from the same transaction). Such anomalies, of deleted items reappearing, have been noted in previous work [38].

In this thesis, we address the problem of *testing* code for correctness against weak behaviors: a developer should be able to write a test that runs their application and then asserts for correct behavior. The main difficulty today is getting coverage of weak behaviors during the test. If one runs the test against the actual production storage system, it is very likely to only result in serializable behaviors because of their optimized implementation. For instance, only 0.0004% of all reads performed on Facebook's TAO storage system were not serializable [71]. Emulators, offered by cloud providers for local development, on the other hand, do not support weaker consistency models at all [8]. Another option, possible when the storage system is available open-source, is to set it up with a tool like Jepsen [63] to inject noise (bring down replicas or delay packets on the network). This approach is unable to provide good coverage at the level of client operations [85] (§4.6). Another line of work has focussed on finding anomalies by identifying non-serializable behavior (§4.7). Anomalies, however, do not always correspond to bugs [25, 52]; they may either not be important (e.g., gather statistics) or may already be handled in the application (e.g., checking and deleting duplicate items).

We present MonkeyDB, a mock in-memory storage system meant for testing correctness of storage-backed applications. MonkeyDB supports common APIs for accessing data (key-value updates, as well as SQL queries), making it an easy substitute for an actual storage system. MonkeyDB can be configured with one of several consistency models. On a read operation, MonkeyDB computes the set of all possible return values allowed under the chosen consistency models, and randomly returns one of them. The developer can then simply execute their test multiple times to get coverage of possible weak behaviors. For the program in Figure 1.4, if we write a test asserting that two sequential reads cannot return empty-cart followed by *f I; I g*, then it takes only 20 runs of the test (on average) to fail the assert. In contrast, the test does not fail when using MySQL with read committed, even after 100k runs.

Design of MonkeyDB MonkeyDB does not rely on stress generation, fault injection, or data replication. Rather, it works directly with a formalization of the given consistency model in order to compute allowed return values.

MonkeyDB implements a *centralized* operational semantics for key-value stores, which is based on the axiomatic definitions of consistency models that we introduced while investigating the algorithmic questions described in Section 1.2. Transactions are executed *serially*, one after another, the concurrency being simulated during the handling of read events. This semantics maintains a history that contains all the past events (from all transactions/sessions), and write events are simply added to the history. The value returned by a read event is established based on a non-deterministic choice of a write-read dependency (concerning this read event) that satisfies the axioms of the considered consistency model. Depending on the weakness of the consistency model, this makes it possible to return values written in arbitrarily "old" transactions, and simulate any concurrent behavior. For instance, the history in Figure 1.4 can be obtained by executing `AddItem`, `DeleteItem`, and then the two reads (serially). The read in `DeleteItem` can take its value from the initial state and "ignore" the previously executed `AddItem`, because the obtained history validates the axioms of causal consistency (or read committed). The same happens for the two later reads in the same session, the first one being able to read from `DeleteItem` and the second one from `AddItem`.

We formally prove that this semantics does indeed simulate any concurrent behavior, by showing that it is equivalent to a semantics where transactions are allowed to interleave. In comparison with concrete implementations, this semantics makes it possible to handle a wide range of consistency models in a uniform way. It only has two sources of non-determinism: the order in which entire transactions are submitted, and the choice of write-read dependencies in read events. This enable better coverage of possible behaviors, the penalty in performance not being an issue in safety testing workloads which are usually small (see our evaluation).

We also extend our semantics to cover SQL queries as well, by compiling SQL queries down to transactions with multiple key-value reads/writes. A table in a relational database is represented using a set of primary key values (identifying uniquely the set of rows) and a set of keys, one for each cell in the table. The set of primary key values is represented using a set of Boolean key-value pairs that simulate its characteristic function (adding or removing an element corresponds to updating one of these keys to `true` or `false`). Then, SQL queries are compiled to read or write accesses to the keys representing a table. For instance, a `SELECT` query that retrieves the set of rows in a table that satisfy a `WHERE` condition is compiled to (1) reading Boolean keys to identify the primary key values of the rows contained in the table, (2) reading keys that represent columns used in the `WHERE` condition, and (3) reading all the keys that represent cells in a row satisfying the `WHERE` condition. This rewriting contains the minimal set of accesses to the cells of a table that are needed to ensure the conventional specification of SQL. It makes it possible to "export" formalizations of key-value store consistency models to SQL transactions.

We present an evaluation of MonkeyDB on several applications, showcasing its superior coverage of weak behaviors as well as bug-finding abilities.

1.4 Thesis Outline

The rest of this dissertation is organized as follows:

- Chapter 2 investigates the problem of testing implementations of CRDTs. It presents formal specifications for such datatypes and studies the asymptotic complexity of checking whether a given execution satisfies a CRDT formal specification.

- Chapter 3 defines axiomatic specifications of several transactional consistency models and establishes complexity results concerning the problem of checking conformance to such specifications for a given execution. It shows that consistency models weaker than Causal Consistency can be checked in polynomial time, while the problem becomes NP-complete for stronger models. In the latter case, it identifies a parameter of executions which enables polynomial-time algorithms when fixed.

- Chapter 4 investigates the testing coverage problem for distributed applications built on top of transactional datastores. It presents the mock in-memory storage system MonkeyDB, which simulates transactional datastores according to their formal specifications. The experimental evaluation of MonkeyDB shows that it provides better test coverage than state of the art setups.

- Chapter 5 concludes and discusses directions for future work.

2 Checking Consistency for Conflict-Free Replicated Data Types

In this chapter we study the tractability of runtime CRDT consistency checking: deciding whether a given execution of a CRDT is consistent with its specification. Our setting captures executions across a set of replicas as per-replica sequences of operations called *histories*. Roughly speaking, a history is *consistent* so long as each operation's return value can be justified according to the operations that its replica has observed so far. In the setting of CRDTs, the determination of a replica's observations is essentially an implementation choice: replicas are only obliged to observe their own operations, and the predecessors of those it has already observed. This relatively-weak constraint on replicas' observations makes the CRDT consistency checking problem unique.

We present logical characterizations of CRDTs, which are built on a notion of *abstract execution*, which relates the operations of a given history with three separate relations: a *read-from* relation, governing the observations from which a given operation constitutes its own return value; a *happens-before* relation, capturing the causal relationships among operations; and a *linearization* relation, capturing any necessary arbitration among non-commutative effects which are executed concurrently, e.g., following a *last-writer-wins* policy. Accordingly, we capture data type specifications with logical axioms interpreted over the read-from, happens-before, and linearization relations of abstract executions, reducing the consistency problem to: does there exist an abstract execution over the given history which satisfies the axioms of the given data type?

We demonstrate the intractability of several replicated data types by reduction from propositional satisfiability (SAT) problems. In particular, we consider the 1-in-3 SAT problem [53], which asks for a truth assignment to the variables of a given set of clauses such that exactly one literal per clause is assigned true. Our reductions essentially simulate the existential choice of a truth assignment with the existential choice of the read-from and happens-before relations of an abstract execution. For a given 1-in-3 SAT instance, we construct a history of replicas obeying carefully-tailored synchronization protocols, which is consistent exactly when the corresponding SAT instance is positive.

Finally, we develop tractable consistency-checking algorithms for individual data types and special cases: replicated growing arrays; multi-value and last-writer-wins registers, when each value is written only once; counters, when replicas are bounded; and sets and flags, when their sizes are also bounded. While the algorithms for each case are tailored to the algebraic properties of the data types they handle, they essentially all function by constructing abstract executions incrementally, processing replicas' operations in prefix order.

The remainder of this chapter is organized as follows:

- Section 2.1 presents logical characterizations of consistency for the replicated register, flag, set, counter, and array data types;
- Section 2.2 introduces reductions from propositional satisfiability problems to consistency checking to demonstrate intractability for replicated flags, sets, counters, and registers; and
- Section 2.3 defines polynomial time consistency-checking algorithms for replicated growable arrays, registers, when written values are unique, counters, when replicas are bounded, and sets and flags, when their sizes are also bounded.

Section 2.4 overviews related work, and Section 2.5 concludes.

2.1 A Logical Characterization of Replicated Data Types

In this section we describe an axiomatic framework for defining the semantics of replicated data types. We consider a set of method names M, and that each method $\mathsf{m} \in \mathsf{M}$ has a number of arguments and a return value sampled from a data domain D. We will use operation labels of the form $\mathsf{m}(a) \stackrel{i}{\Rightarrow} b$ to represent the call of a method $\mathsf{m} \in \mathsf{M}$, with argument $a \in \mathsf{D}$, and resulting in the value $b \in \mathsf{D}$. Since there might be multiple calls to the same method with the same arguments and result, labels are tagged with a unique identifier i. We will ignore identifiers when unambiguous.

The interaction between a data type implementation and a client is represented by a *history* $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ which consists of a set of operation labels Op and a partial *replica order* ro ordering operations issued by the client on the same replica. Usually, ro is a union of sequences, each sequence representing the operations issued on the same replica, and the *width* of ro, i.e., the maximum number of mutually-unordered operations, gives the number of replicas in a given history.

To characterize the set of histories $h = \langle \mathsf{Op}, \mathsf{ro} \rangle$ admitted by a certain replicated data type, we use *abstract executions* $e = \langle \mathsf{rf}, \mathsf{hb}, \mathsf{lin} \rangle$, which include:

- a *read-from* binary relation rf over operations in Op, which identifies the set of updates needed to "explain" a certain return value, e.g., a write operation explaining the return value of a read,
- a strict partial *happens-before* order hb, which includes ro and rf, representing the causality constraints in an execution, and
- a strict total *linearization* order lin, which includes hb, used to model conflict resolution policies based on timestamps.

In this work, we consider replicated data types which satisfy *causal consistency* [68], i.e., updates which are related by cause and effect relations are observed by all replicas in the same order. This follows from the fact that the happens-before order is constrained to be a partial order, and thus transitive (other forms of weak consistency don't pose this constraint). Some of the replicated data types we consider in this work do *not* consider resolution policies based on timestamps and in those cases, the linearization order can be ignored.

ReadFrom(R)
$$\forall o_1, o_2.\ \mathsf{rf}(o_1, o_2) \Rightarrow R(o_1, o_2)$$

ReadFromMaximal(R)
$$\forall o_1, o_2, o_3.\ \mathsf{rf}(o_1, o_2) \wedge R(o_3, o_2) \Rightarrow \neg \mathsf{hb}(o_1, o_3) \vee \neg \mathsf{hb}(o_3, o_2)$$

ReadAllMaximals(R)
$$\forall o_1, o_2.\ \mathsf{hb}(o_1, o_2) \wedge R(o_1, o_2) \Rightarrow \exists o_3.\ \mathsf{hb}(o_1, o_3) \wedge \mathsf{rf}(o_3, o_2)$$

ClosedRF(R)
$$\forall o_1, o_2, o_3.\ R(o_1, o_2) \wedge \mathsf{hb}(o_1, o_3) \wedge \mathsf{rf}(o_3, o_2) \Rightarrow \mathsf{rf}(o_1, o_2)$$

RetvalSet(X, v, Y)
$$\forall o_1.\ \mathsf{meth}(o_1) = X \wedge \mathsf{ret}(o_1) = v \Leftrightarrow \exists o_2.\ \mathsf{rf}(o_2, o_1) \wedge \mathsf{meth}(o_2) = Y \wedge \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$$

RetvalCounter
$$\forall o_1.\ \mathsf{meth}(o_1) = \mathsf{read} \Rightarrow \mathsf{ret}(o_1) = |\{o_2 : \mathsf{meth}(o_2) = \mathsf{inc} \wedge \mathsf{rf}(o_2, o_1)\}| - |\{o_2 : \mathsf{meth}(o_2) = \mathsf{dec} \wedge \mathsf{rf}(o_2, o_1)\}|$$

LinLWW
$$\forall o_1, o_2, o_3.\ \mathsf{rf}(o_1, o_2) \wedge \mathsf{meth}(o_3) = \mathsf{write} \wedge \mathsf{arg}_1(o_3) = \mathsf{arg}(o_2) \wedge \mathsf{hb}(o_3, o_2) \Rightarrow \mathsf{lin}(o_3, o_1)$$

RetvalReg
$$\forall o_1, v.\ \mathsf{meth}(o_1) = \mathsf{read} \wedge v \in \mathsf{ret}(o_1) \Rightarrow \exists! o_2.\ \mathsf{rf}(o_2, o_1) \wedge \mathsf{meth}(o_2) = \mathsf{write} \wedge \mathsf{arg}_2(o_2) = v$$

Figure 2.1: The axiomatic semantics of replicated data types. Quantified variables are implicitly distinct, and $\exists! o$ denotes the existence of a unique operation o.

A *replicated data type* is defined by a set of first-order axioms Φ characterizing the relations in an abstract execution. A history h is *admitted* by a data type when there exists an abstract execution e such that $\langle h, e\rangle \models \Phi$. The satisfaction relation $\models$ is defined as usual in first order logic. The *admissibility problem* is the problem of checking whether a history h is admitted by a given data type.

In the following, we define the replicated data types with respect to which we study the complexity of the admissibility problem. The axioms used to define them are listed in Figure 2.1 and Figure 2.2. These axioms use the function symbols meth-od, arg-ument, and ret-urn interpreted over operation labels, whose semantics is self-explanatory.

2.1.1 Replicated Sets and Flags

The Add-Wins Set and Remove-Wins Set [88] are two implementations of a replicated set with operations $\mathsf{add}(x)$, $\mathsf{remove}(x)$, and $\mathsf{contains}(x)$ for adding, removing, and checking membership of an element x. Although the meaning of these methods is self-evident from their names, the result of conflicting concurrent operations is not evident. When concurrent $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ operations are delivered to a certain replica, the Add-Wins Set chooses to keep the element x in the set, so every subsequent invocation of $\mathsf{contains}(x)$ on this replica returns $true$, while the Remove-Wins Set makes the dual choice of removing x from the set.

The formal definition of their semantics uses abstract executions where the read-from relation associates sets of $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ operations to $\mathsf{contains}(x)$ operations. Therefore, the predicate $\mathsf{ReadOk}(o_1, o_2)$ is defined by

$$\mathsf{meth}(o_1) \in \{\mathsf{add}, \mathsf{remove}\} \wedge \mathsf{meth}(o_2) = \mathsf{contains} \wedge \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$$

and the Add-Wins Set is defined by the following set of axioms:

ReadFrom(ReadOk) ^ ReadFromMaximal(ReadOk) ^
ReadAllMaximals(ReadOk) ^ RetvalSet(contains; *true*; add)

ReadFromMaximal says that every operation read by a contains(x) is maximal among its hb-predecessors that add or remove x while ReadAllMaximals says that all such maximal hb-predecessors are read. The RetvalSet instantiation ensures that a contains(x) returns *true* iff it reads-from at least one add(x).

The definition of the Remove-Wins Set is similar, except for the parameters of RetvalSet, which become RetvalSet(contains; *false*; remove); i.e., a contains(x) returns *false* iff it reads-from at least one remove(x).

The Enable-Wins Flag and Disable-Wins Flag are implementations of a set of flags with operations: enable(x), disable(x), and read(x), where enable(x) turns the flag x to true, disable(x) turns x to false, while read(x) returns the state of the flag x. Their semantics is similar to the Add-Wins Set and Remove-Wins Set, respectively, where enable(x), disable(x), and read(x) play the role of add(x), remove(x), and contains(x), respectively. Their axioms are defined as above.

2.1.2 Replicated Registers

We consider two variations of replicated registers called Multi-Value Register (MVR) and Last-Writer-Wins Register (LWW) [88] which maintain a set of registers and provide write(x,v) operations for writing a value v on a register x and read(x) operations for reading the content of a register x (the domain of values is kept unspecified since it is irrelevant). While a read(x) operation of MVR returns *all* the values written by concurrent writes which are maximal among its happens-before predecessors, therefore, leaving the responsibility for solving conflicts between concurrent writes to the client, a read(x) operation of LWW returns a single value chosen using a conflict-resolution policy based on timestamps. Each written value is associated to a timestamp, and a read operation returns the most recent value w.r.t. the timestamps. This order between timestamps is modeled using the linearization order of an abstract execution.

Therefore, the predicate ReadOk(o_1; o_2) is defined by

$$\mathrm{meth}(o_1) = \mathrm{write} \wedge \mathrm{meth}(o_2) = \mathrm{read} \wedge \mathrm{arg}_1(o_1) = \mathrm{arg}(o_2) \wedge \mathrm{arg}_2(o_1) \in \mathrm{ret}(o_2)$$

(we use $\mathrm{arg}_1(o_1)$ to denote the first argument of a write operation, i.e., the register name, and $\mathrm{arg}_2(o_1)$ to denote its second argument, i.e., the written value) and the MVR is defined by the following set of axioms:

ReadFrom(ReadOk) ^ ReadFromMaximal(ReadOk) ^
ReadAllMaximals(ReadOk) ^ RetvalReg

where RetvalReg ensures that a read(x) operation reads from a write(x,v) operation, for each value v in the set of returned values [1].

[1] For simplicity, we assume that every history contains a set of write operations writing the initial values of variables, which precede every other operation in replica order.

<u>ReadFromRGA</u>

$$\begin{array}{l}\forall o_2.\ \mathsf{meth}(o_2) = \mathsf{addAfter} \Rightarrow \mathsf{arg}_1(o_2) = \quad \lor \\ \qquad\qquad \exists o_1.\ \mathsf{meth}(o_1) = \mathsf{addAfter} \land \mathsf{arg}_2(o_1) = \mathsf{arg}_1(o_2) \land \mathsf{rf}(o_1, o_2) \\ \quad \land\ \mathsf{meth}(o_2) = \mathsf{remove} \Rightarrow \exists o_1.\ \mathsf{meth}(o_1) = \mathsf{addAfter} \land \mathsf{arg}_2(o_1) = \mathsf{arg}(o_2) \land \mathsf{rf}(o_1, o_2) \\ \quad \land\ \mathsf{meth}(o_2) = \mathsf{read} \Rightarrow \forall v \in \mathsf{ret}(o_2)\ \exists o_1.\mathsf{meth}(o_1) = \mathsf{addAfter} \land \mathsf{arg}_2(o_1) = v \land \mathsf{rf}(o_1, o_2)\end{array}$$

<u>RetvalRGA</u>

$$\begin{array}{l}\forall o_1, o_2.\ \mathsf{meth}(o_1) = \mathsf{read} \land \mathsf{meth}(o_2) = \mathsf{addAfter} \land \mathsf{hb}(o_2, o_1) \land \mathsf{arg}_2(o_2) \notin \mathsf{ret}(o_1) \\ \qquad\qquad \Rightarrow \exists o_3.\ \mathsf{meth}(o_3) = \mathsf{remove} \land \mathsf{arg}(o_3) = \mathsf{arg}_2(o_2) \land \mathsf{rf}(o_3, o_1)\end{array}$$

<u>LinRGA</u>

$$\begin{array}{l}\forall o_1, o_2.\ (\mathsf{meth}(o_1) = \mathsf{meth}(o_2) = \mathsf{addAfter} \land \mathsf{arg}_1(o_1) = \mathsf{arg}_1(o_2) \land \\ \qquad \exists o_3, o_4, o_5.\ \mathsf{meth}(o_3) = \mathsf{meth}(o_4) = \mathsf{addAfter} \land \mathsf{rf}_{\mathsf{addAfter}}(o_1, o_3) \land \mathsf{rf}_{\mathsf{addAfter}}(o_2, o_4) \land \\ \qquad \mathsf{meth}(o_5) = \mathsf{read} \land \mathsf{arg}_2(o_4) <_{o_5} \mathsf{arg}_2(o_3)) \Rightarrow \mathsf{lin}(o_1, o_2)\end{array}$$

Figure 2.2: Axioms used to define the semantics of RGA.

LWW is obtained from the definition of MVR by replacing ReadAllMaximals with the axiom LinLWW which ensures that every $\mathsf{write}(x,\_)$ operation which happens-before a $\mathsf{read}(x)$ operation is linearized before the $\mathsf{write}(x,\_)$ operation from where the $\mathsf{read}(x)$ takes its value (when these two write operations are different). This definition of LWW is inspired by the "bad-pattern" characterization in [21], corresponding to their causal convergence criterion.

2.1.3 Replicated Counters

The replicated counter datatype [88] maintains a set of counters interpreted as integers (the counters can become negative). This datatype provides operations $\mathsf{inc}(x)$ and $\mathsf{dec}(x)$ for incrementing and decrementing a counter x, and $\mathsf{read}(x)$ operations to read the value of the counter x. The semantics of the replicated counter is quite standard: a $\mathsf{read}(x)$ operation returns the value computed as the difference between the number of $\mathsf{inc}(x)$ operations and $\mathsf{dec}(x)$ operations among its happens-before predecessors. The axioms defined below will enforce the fact that a $\mathsf{read}(x)$ operation reads-from all its happens-before predecessors which are $\mathsf{inc}(x)$ or $\mathsf{dec}(x)$ operations.

Therefore, the predicate $\mathsf{ReadOk}(o_1, o_2)$ is defined by

$$\mathsf{meth}(o_1) \in \{\mathsf{inc}, \mathsf{dec}\} \land \mathsf{meth}(o_2) = \mathsf{read} \land \mathsf{arg}(o_1) = \mathsf{arg}(o_2)$$

and the replicated counter is defined by the following set of axioms:

$$\textsc{ReadFrom}(\mathsf{ReadOk}) \land \textsc{ClosedRF}(\mathsf{ReadOk}) \land \textsc{RetvalCounter}.$$

2.1.4 Replicated Growable Array

The Replicated Growing Array (RGA) [86] is a replicated list used for text-editing applications. RGA supports three operations: $\mathsf{addAfter}(a,b)$ which adds the character b immediately after the occurrence of the character a assumed to be present in the list, $\mathsf{remove}(a)$ which removes a assumed to be present in the list, and $\mathsf{read}()$ which returns the list contents. It is assumed that a character is added at most once [2]. The conflicts between concurrent addAfter operations that add a character immediately after the same character is solved using timestamps (i.e., each added character is associated to a timestamp and the order between characters depends on the order between the corresponding timestamps), which in the axioms below are modeled by the linearization order.

Figure 2.2 lists the axioms defining RGA. ReadFromRGA ensures that:

- every addAfter(a,b) operation reads-from the addAfter(_,a) adding the character a, except when $a =$ which denotes the "root" element of the list[3],
- every remove(a) operation reads-from the operation adding a, and
- every read operation returning a list containing a reads-from the operation addAfter(_,a) adding a.

Then, RetvalRGA ensures that a read operation o_1 happening-after an operation adding a character a reads-from a remove(a) operation when a doesn't occur in the list returned by o_1 (the history must contain a remove(a) operation because otherwise, a should have occurred in the list returned by the read).

Finally, LinRGA models the conflict resolution policy by constraining the linearization order between addAfter(a,_) operations adding some character immediately after the same character a. As a particular case, LinRGA enforces that addAfter(a,b) is linearized before addAfter(a,c) when a read operation returns a list where c precedes b (addAfter(a,b) results in the list $a \quad b$ and applying addAfter(a,c) on $a \quad b$ results in the list $a \quad c \quad b$). However, this is not sufficient: assume that the history contains the two operations addAfter(a,b) and addAfter(a,c) along with two operations remove(b) and addAfter(b,d). Then, a read operation returning the list $a \quad c \quad d$ must enforce that addAfter(a,b) is linearized before addAfter(a,c) because this is the only order between these two operations that can lead to the result $a \quad c \quad d$, i.e., executing addAfter(a,b), addAfter(b,d), remove(b), addAfter(a,c) in this order. LinRGA deals with any scenario where arbitrarily-many characters can be removed from the list: $\mathsf{rf}_{\mathsf{addAfter}}$ is the reflexive and transitive closure of the projection of rf on addAfter operations and $<_{o_5}$ denotes the order between characters in the list returned by the read operation o_5.

2.2 Intractability for Registers, Sets, Flags, and Counters

In this section, we demonstrate that checking the consistency is intractable for many widely-used data types. While this is not completely unexpected, since some related consistency-checking problems like sequential consistency are also intractable [54], this contrasts recent tractability results for checking strong consistency (i.e., linearizability) of common non-replicated data types like sets, maps, and queues [43]. In fact, in many cases, we show that intractability even holds if the number of replicas is fixed.

Our proofs of intractability follow the general structure of Gibbons and Korach's proofs for the intractability of checking sequential consistency (SC) for atomic registers with read and write

[2] In a practical context, this can be enforced by tagging characters with replica identifiers and sequence numbers.
[3] This element is not returned by read operations.

	Replica 0	Replica 1	Replica 2
Round 0	Enable(x_1)	Disable(x_1)	
	…	…	
	Enable(x_n)	Disable(x_n)	
Barrier 1	Enable(y_0)	Enable(y_1)	Enable(y_2)
	Read(y_1) = *true*	Read(y_0) = *true*	Read(y_0) = *true*
	Read(y_2) = *true*	Read(y_2) = *true*	Read(y_1) = *true*
Round 1	Read($_1$) = *true*	Read($_1$) = *true*	Read($_1$) = *true*
	Read($_1$) = *false*	Read($_1$) = *false*	Read($_1$) = *false*
	Read($_1$) = *false*	Read($_1$) = *false*	Read($_1$) = *false*
	Disable($_1$)	Disable($_1$)	Disable($_1$)
	Enable($_1$)	Enable($_1$)	Enable($_1$)
Barrier 2	Disable(y_0)	Disable(y_1)	Disable(y_2)
	Read(y_1) = *false*	Read(y_0) = *false*	Read(y_0) = *false*
	Read(y_2) = *false*	Read(y_2) = *false*	Read(y_1) = *false*
	…	…	…
Round m	Read($_m$) = *true*	Read($_m$) = *true*	Read($_m$) = *true*
	Read($_m$) = *false*	Read($_m$) = *false*	Read($_m$) = *false*
	Read($_m$) = *false*	Read($_m$) = *false*	Read($_m$) = *false*
	Disable($_m$)	Disable($_m$)	Disable($_m$)
	Enable($_m$)	Enable($_m$)	Enable($_m$)

Figure 2.3: The encoding of a 1-in-3 SAT problem $\bigvee_{i=1}^{m}(\ _i \_ \ _i \_ \ _i)$ over variables $x_1, \ldots, x_n$ as a 3-replica history of a flag data type. Besides the flag variable x_j for each propositional variable x_j, the encoding adds per-replica variables y_j for synchronization barriers.

operations [54]. In particular, we reduce a specialized type of NP-hard propositional satisfiability (SAT) problem to checking whether histories are admitted by a given data type. While our construction borrows from Gibbons and Korach's, the adaptation from SC to CRDT consistency requires a significant extension to handle the consistency relaxation represented by abstract executions: rather than a direct sequencing of threads' operations, CRDT consistency requires the construction of three separate relations: read-from, happens-before, and linearization.

Technically, our reductions start from the 1-in-3 SAT problem [53]: given a propositional formula $\bigvee_{i=1}^{m}(\ _i \_ \ _i \_ \ _i)$ over variables $x_1, \ldots, x_n$ with only positive literals, i.e., $\ _i, \ _i, \ _i \in \{x_1, \ldots, x_n\}$, does there exist an assignment to the variables such that exactly one of $\ _i, \ _i, \ _i$ per clause is assigned $true$? The proofs of Theorems 2.2.1 and 2.2.2 reduce 1-in-3 SAT to CRDT consistency checking.

Theorem 2.2.1. *The admissibility problem is NP-hard when the number of replicas is fixed for the following data types: Add-Wins Set, Remove-Wins Set, Enable-Wins Flag, Disable-Wins Flag, Multi-Value Register, and Last-Writer-Wins Register.*

Proof. We demonstrate a reduction from the 1-in-3 SAT problem. For a given problem $p = \bigvee_{i=1}^{m}(\ _i \_ \ _i \_ \ _i)$ over variables $x_1, \ldots, x_n$, we construct a 3-replica history h_p of the flag data type — either enable- or disable-wins — as illustrated in Figure 2.3. The encoding includes a flag variable x_j for each propositional variable x_j, along with a per-replica flag variable y_j used to implement synchronization barriers. Intuitively, executions of h_p proceed in $m + 1$ rounds: the first round corresponds to the assignment of a truth valuation, while subsequent rounds check the validity of each clause given the assignment. The reductions to sets and registers are slight variations on this proof, in which the Read, Enable, and Disable operations are replaced with Contains, Add, and Remove, respectively, and Read and Writes of values 1 and 0, respectively.

It suffices to show that the constructed history h_p is admitted if and only if the given problem p is satisfiable (it is easy to see that the size of h_p is linear in the size of p, and that h_p can be computed in linear time). Since the flag data type does not constrain the linearization relation of its abstract executions, we regard only the read-from and happens-before components. The construction of h_p ensures the happens-before relations of its abstract executions:

1. does not interleave operations from different rounds. Each consecutive rounds are separated by the barriers in happens-before relations; and

2. at each round, only one replica, say replica i, can finish its Reads then finish its Enables/Disables, then $(i+1)\ \mathrm{mod}\ 3$ replica can finish its Reads and so on. And these Enables/Disables from one round are totally ordered between replicas by the happens-before relation.

In other words, replicas appear to execute atomically per round, in a round-robin fashion. Furthermore, since all operations in a given round happen before the operations of subsequent rounds, the values of flag variables are consistent across rounds — i.e., as read by the first replica to execute in a given round — and determined in the initial round either by conflict resolution — i.e., enable- or disable-wins — or by happens-before, in case of conflict resolution would have been inconsistent with subsequent reads.

The correctness of the construction is stated in the following lemma:

Lemma 2.2.1. $p = \bigvee_{i=1}^{m}(\ {}_i \_ \ {}_i \_ \ {}_i)$ *is satisfiable if and only if* h_p *is admissible*

Proof. (Only-if direction). Assume that $\bigvee_{i=1}^{1}(\ {}_i \_ \ {}_i \_ \ {}_i)$ is satisfiable, i.e., there exists a variable assignment for which each clause has exactly one literal interpreted as true.

We construct a happens-before relation hb such that: (1) if $(x_i) = false$, then $Enable(x_i)$ in Replica 0 is visible to $Disable(x_i)$ in Replica 1, *i.e.* $(Enable(x_i); Disable(x_i))\ 2\ \mathsf{hb}$ (this ensures that the value of x_i is $false$ after Barrier 1), and similarly, (2) if $(x_i) = true$, then $(Disable(x_i); Enable(x_i))\ 2\ \mathsf{hb}$. Note this does not introduce any cycle in hb because x_is are Enabled and Disabled in the same order in Replica 0 and Replica 1.

Then, for each barrier i, all the Enable operations happen-before all the Read operations. Also, for each round i, if ${}_i$ is true in clause i (w.r.t.), then we make all the operations of the round i at replica 0 happen-before operations of the round i at replica 1, and all the operations of the round i at replica 1 happen-before operations of the round i at replica 2. This makes the history admissible because if ${}_1$ is true, then ${}_1$ and ${}_1$ are false. So the reads of round i at replica 0 are correct. Then the updates of round i at replica 0 make ${}_1$ false and ${}_1$ true (${}_1$ remains false). So now, the reads of round i at replica 1 are also correct. The same reasoning can be applied for the reads at replica 2. The cases ${}_i$ true or ${}_i$ true lead to a similar definition of happens-before, ordering replicas in the order 1, 2, 0 if ${}_i$ is true, and 2, 0, 1, if ${}_1$ is true. A straightforward proof by induction allows to prove that the history is admissible w.r.t. a happens-before relation defined in this manner.

(If direction). Assume that h_p is admissible. We give a series of lemmas that characterize the happens-before (read-from) relation of h_p.

Lemma 2.2.2. *The Reads of each barrier read-from the Enables or Disables of the same barrier.*

Proof. We give a proof by induction on the number of the barrier.
(Base case). Note that only replica i Enables or Disables y_i. Assume by contradiction that at barrier 1, $\mathsf{Read}(y_j) = true$ at replica i reads from an operation in a barrier other than 1. Then, it must read from an operation of barrier 3 at least (y_j is set to true at odd numbered barriers). Now at barrier 2 and replica j, $\mathsf{Read}(y_i) = false$ reads from a Disable because replica j has $\mathsf{Read}(y_i) = true$ at barrier 1. So at barrier 2 and replica j, $\mathsf{Read}(y_i) = false$ can read from a Disable which is at even numbered barriers, at least from barrier 2. This defines a cycle in the happens-before order, which contradicts the admissibility of h_p:

- $\mathsf{Enable}(y_j)$ at barrier 3 and replica j happens-before $\mathsf{Read}(y_j) = true$ at barrier 1 and replica i because of read-from,
- $\mathsf{Read}(y_j) = true$ at barrier 1 and replica i happens-before $\mathsf{Disable}(y_i)$ at barrier 2 and replica i,
- $\mathsf{Disable}(y_i)$ at barrier 2 and replica i happens-before $\mathsf{Read}(y_i) = false$ at barrier 2 and replica j because of read-form, and
- $\mathsf{Read}(y_i) = false$ at barrier 2 and replica j happens-before $\mathsf{Enable}(y_j)$ at barrier 3 and replica j.

(Induction step). By the induction hypothesis, barrier k always reads from barrier k itself. Therefore, $\mathsf{Read}(y_j)$ of barrier $(k + 1)$ happens after the update of y_j from barrier k. Without loss of generality, let us assume that barrier k contains Enable operations. Since barrier k contains Enable operations, the $false$ reads in barrier $(k + 1)$ must read from a barrier strictly greater than k. Using the same logic from the base case, this would imply a cycle in the happens-before relation. □

Lemma 2.2.2 ensures that all the operations from all replicas before barrier i happen before every operation (from any replica) after barrier i.

We say that two Read operations *see the same value of* x_i when one Read reads-from an $\mathsf{Enable}(x_i)$ if and only if the other Read also reads-from an $\mathsf{Enable}(x_i)$ (these Enable operations may not be the same). Also, a read in a round k is called *initial* if it does not happen before an Enable or Disable from the same round.

Lemma 2.2.3. *Initial Reads of two consecutive rounds see the same value of* x_i*, for each* i*.*

Proof. Note that the Reads from each round read only from updates from the same round or from preceding rounds. Reading from any later round is not possible, because, by Lemma 2.2.2, that will introduce a happen-before cycle between the current round and that later round. Also, in each round, there exist one replica which does not read-from other replicas in the same round. If replica p is reading from replica q, then replica q again can not read from replica p, because it will create a cycle in hb between replica p and q in the same round. So replica q has to read from replica r. But then replica r will have to read from replica p, which creates a cycle between replica p, q, and r in the same round.

So there exists one replica, which reads-from updates till last round. Since the Reads of that replica are successful, it ensures only one of $\ _k; \ _k; \ _k$ are true *i.e.* only one of those Reads reads-from an Enable. Hence, the first true Reads at other replicas must read-from the updates in the same round. Therefore, all the operations from a round $k \quad 1$ are totally ordered w.r.t. the happens-before, all operations in one replica before all operations in another. That is, if replica 0 was the first one to finish its reads, $\mathsf{Read}(\ _k) = true$ reads-from $\mathsf{Enable}(\ _k)$ and $\mathsf{Read}(\ _k) = true$ reads-from $\mathsf{Enable}(\ _k)$. Since the updates are totally ordered and they only flip the read values of x_i twice, *i.e.* if the first read on x_i is $false$, then it does not read-from any Enable till round $(k \quad 1)$ and at round k, after $\mathsf{Read}(x_i) = false$, $\mathsf{Enable}(x_i)$ and $\mathsf{Disable}(x_i)$ are ordered by hb. So the hb-maximal update on x_i on round k stays $\mathsf{Disable}(x_i)$. Similarly we can show, the hb-maximal update on x_i on round k stays $\mathsf{Enable}(x_i)$ when the first $\mathsf{Read}(x_i)$ was true.

When round $(k + 1)$ begins, because of Lemma 2.2.2, it "sees" all the updates at the end of round k, which includes the updates from earlier rounds.

- If x_i is not modified in round k, then round $(k + 1)$ will read-from from the same update for x_i as round k.
- If x_i is modified in round k, any hb-maximal $\mathsf{Read}(x_i)$ at round $(k + 1)$ will read-form hb-maximal updates at round k by lemma 2.2.2. And, the hb-maximal update on x_i at the end of round k stays the same as the update which round k read-form at the beginning.

□

Going back to the proof of Lemma 2.2.1, p is satisfiable using an assignment defined by the initial Reads of each round (which see the same values by Lemma 2.2.3). This assignment satisfies the 1-in-3 SAT formula p because at each round, there is a replica that happens-before operations in the same round at the other replicas, and the Reads of that replica see exactly one flag as true.
□

□

Theorem 2.2.1 establishes intractability of consistency for the aforementioned sets, flags, and registers, independently from the number of replicas. In contrast, our proof of Theorem 2.2.2 for counter data types depends on the number of replicas, since our encoding requires two replicas per propositional variable. Intuitively, since counter increments and decrements are commutative, the initial round in the previous encoding would have fixed all counter values to zero. Instead, the next encoding isolates initial increments and decrements to independent replicas.

Theorem 2.2.2. *The admissibility problem for the Counter data type is NP-hard.*

We demonstrate a reduction from the 1-in-3 SAT problem. For a given problem $p = \bigvee_{i=1}^{m}(\ _i \vee \ _i \vee \ _i)$ over variables $x_1, \ldots, x_n$, we construct a history h_p of the counter data type over $2n + 3$ replicas, as illustrated in Figure 2.4.

Besides the differences imposed due to the commutativity of counter increments and decrements, our reduction follows the same strategy as in the proof of Theorem 2.2.1: the happens-before relation of h_p's abstract executions order every pair of operations in distinct rounds (of

	Replica 0	Replica $2j+1$	Replica $2j+2$
Round 0		$\mathsf{Inc}(y)$ $\mathsf{Inc}(x_j)$	$\mathsf{Inc}(y)$ $\mathsf{Dec}(x_j)$
	$\mathsf{Read}(y) = n$		
		Replica 1	Replica 2
Barrier 1	$\mathsf{Inc}(y_0)$ $\mathsf{Read}(y_1) = 1$ $\mathsf{Read}(y_2) = 1$	$\mathsf{Inc}(y_1)$ $\mathsf{Read}(y_2) = 1$ $\mathsf{Read}(y_0) = 1$	$\mathsf{Inc}(y_2)$ $\mathsf{Read}(y_0) = 1$ $\mathsf{Read}(y_1) = 1$
Round 1	$\mathsf{Read}(\ _1) = 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Dec}(\ _1); \mathsf{Dec}(\ _1)$ $\mathsf{Inc}(\ _1); \mathsf{Inc}(\ _1)$	$\mathsf{Read}(\ _1) = 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Dec}(\ _1); \mathsf{Dec}(\ _1)$ $\mathsf{Inc}(\ _1); \mathsf{Inc}(\ _1)$	$\mathsf{Read}(\ _1) = 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Read}(\ _1) = \ 1$ $\mathsf{Dec}(\ _1); \mathsf{Dec}(\ _1)$ $\mathsf{Inc}(\ _1); \mathsf{Inc}(\ _1)$
Barrier 2	$\mathsf{Dec}(y_0)$ $\mathsf{Read}(y_1) = 0$ $\mathsf{Read}(y_2) = 0$	$\mathsf{Dec}(y_1)$ $\mathsf{Read}(y_2) = 0$ $\mathsf{Read}(y_0) = 0$	$\mathsf{Dec}(y_2)$ $\mathsf{Read}(y_0) = 0$ $\mathsf{Read}(y_1) = 0$
	...	...	...
Round m	$\mathsf{Read}(\ _m) = 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Dec}(\ _m); \mathsf{Dec}(\ _m)$ $\mathsf{Inc}(\ _m); \mathsf{Inc}(\ _m)$	$\mathsf{Read}(\ _m) = 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Dec}(\ _m); \mathsf{Dec}(\ _m)$ $\mathsf{Inc}(\ _m); \mathsf{Inc}(\ _m)$	$\mathsf{Read}(\ _m) = 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Read}(\ _m) = \ 1$ $\mathsf{Dec}(\ _m); \mathsf{Dec}(\ _m)$ $\mathsf{Inc}(\ _m); \mathsf{Inc}(\ _m)$
Barrier $m+1$	$\mathsf{Inc}(y_0)$ or $\mathsf{Dec}(y_0)$ $\mathsf{Read}(y_1) = 1$ or 0 $\mathsf{Read}(y_2) = 1$ or 0	$\mathsf{Inc}(y_1)$ or $\mathsf{Dec}(y_1)$ $\mathsf{Read}(y_2) = 1$ or 0 $\mathsf{Read}(y_0) = 1$ or 0	$\mathsf{Inc}(y_2)$ or $\mathsf{Dec}(y_2)$ $\mathsf{Read}(y_0) = 1$ or 0 $\mathsf{Read}(y_1) = 1$ or 0
Round $m+1$	$\mathsf{Read}(y) = n$		

Figure 2.4: The encoding of a 1-in-3 SAT problem $\bigvee_{i=1}^{m}(\ _i \_\ \ _i \_\ \ _i)$ over variables $x_1, \ldots, x_n$ as the history of a counter over $2n + 3$ replicas. Besides the counter variables x_j encoding propositional variables x_j, the encoding adds a variable y encoding the number of initial increments and decrements, and a variable z to implement synchronization barriers.

Replicas 0–2), and every operation in a given (non-initial) round. As before, Replicas 0–2 appear to execute atomically per round, in a round-robin fashion, and counter variables are consistent across rounds. The key difference is that here abstract executions' happens-before relations only relate the operations of either Replica $2j+1$ or $2j+2$, for each $j = 1, \ldots, n$, to operations in subsequent rounds: the other's operations are never observed by other replicas. Our encoding ensures that exactly one of each is observed by ensuring that the counter y is incremented exactly n times — and relying on the fact that every variable appears in some clause so that a read that observed neither or both would yield the value zero, which is inconsistent with h_p. Otherwise, our reasoning follows the proof of Theorem 2.2.1, in which the read-from relation selects all increments and decrements of the same counter variable in happens-before order.

2.3 Polynomial-Time Algorithms

2.3.1 Registers and Arrays

We show that the problem of checking consistency is polynomial time for RGA, and even for LWW and MVR under the assumption that each value is written at most once, i.e., for each value

```
Input: A differentiated history h = hOp; roi and a datatype T.
Output: true iff h satisfies the axioms of T.
rf    ComputeRF(h,ReadFrom[T],RetVal[T] );
if rf = ? then return false;
hb    (ro [ rf)+;
if hb is cyclic or hh; rf; hbi 6|= ReadFromMaximal[T] ^ ReadAllMaximals[T] then
  | return false;
lin    hb;
lin    LinClosure(hb,Lin[T]);
if lin is cyclic then return false;
return true;
```

Algorithm 1: Consistency checking for RGA, LWW, and MVR. Re…[T] refers to an axiom of T, or *true* when T lacks such an axiom. The relation R^+ denotes the transitive closure of R.

v, the input history contains at most one write operation write(x,v). Histories satisfying this assumption are called *differentiated*. The latter is a restriction motivated by the fact that practical implementations of these datatypes are data-independent [96], i.e., their behavior doesn't depend on the concrete values read or written and any potential buggy behavior can be exposed in executions where each value is written at most once. Also, in a testing environment, this restriction can be enforced by tagging each value with a replica identifier and a sequence number.

In all three cases, the feature that enables polynomial time consistency checking is the fact that the read-from relation becomes fixed for a given history, i.e., if the history is consistent, then there exists exactly one read-from relation rf that satisfies the ReadFrom_ and RetVal_ axioms, and rf can be derived syntactically from the operation labels (using those axioms). Then, our axiomatic characterizations enable a consistency checking algorithm which roughly, consists in instantiating those axioms in order to compute an abstract execution.

The consistency checking algorithm for RGA, LWW, and MVR is listed in Algorithm 1. It computes the three relations rf, hb, and lin of an abstract execution using the datatype's axioms. The history is declared consistent iff there exist satisfying rf and hb relations, and the relations hb and lin computed this way are acyclic. The acyclicity requirement comes from the definition of abstract executions where hb and lin are required to be partial/total orders. While an abstract execution would require that lin is a total order, this algorithm computes a partial linearization order. However, any total order compatible with this partial linearization would satisfy the axioms of the datatype.

ComputeRF computes the read-from relation rf satisfying the ReadFrom_ and RetVal_ axioms. In the case of LWW and MVR, it defines rf as the set of all pairs formed of write(x,v) and read(x) operations where v belongs to the return value of the read. By RetVal_, each read(x) operation must be associated to at least one write(x,_) operation. Also, the fact that each value is written at most once implies that this rf relation is uniquely defined, e.g., for LWW, it is not possible to find two write operations that could be rf related to the same read operation. In general, if there exists no rf relation satisfying these axioms, then ComputeRF returns a distinguished value ? to signal a consistency violation. Note that the computation of the read-from for LWW and MVR is quadratic time[4] since the constraints imposed by the axioms relate only to the op-

[4] Assuming constant time lookup/insert operations (e.g., using hashmaps), this complexity is linear time.

Input: A history $h = \langle Op, ro \rangle$ of RGA.
Output: An rf satisfying ReadFromRGA $\wedge$ RetvalRGA, if exists; $\bot$ o/w

$rf \leftarrow \{(o_1, o_2) : meth(o_1) = addAfter, meth(o_2) \in \{addAfter, remove, read\}, arg_2(o_1) = arg_1(o_2) \vee arg_2(o_1) \in ret(o_2)\}$;
if $\langle h, rf \rangle \not\models$ ReadFromRGA **then return** $\bot$;
while *true* **do**
  $rf_1 \leftarrow \emptyset$;
  foreach $o_1, o_2 \in Op$ s.t. $\langle o_2, o_1 \rangle \in (rf \cup ro)^+$ and $meth(o_1) = read$ and $meth(o_2) = addAfter$ and $arg_2(o_2) \notin ret(o_1)$ **do**
    if $\exists o_3 \in Op$ s.t. $meth(o_3) = remove$ and $arg(o_3) = arg_2(o_2)$ **then**
      $rf_1 \leftarrow rf_1 \cup \{\langle o_3, o_1 \rangle\}$;
    else
      return $\bot$;
  if $rf_1 \subseteq rf$ **then break**;
  else $rf \leftarrow rf \cup rf_1$;
return rf;

Algorithm 2: The procedure ComputeRF for RGA.

eration labels, the methods they invoke or their arguments. The case of RGA is slightly more involved because the axiom RetvalRGA introduces more read-from constraints based on the happens-before order which includes ro and the rf itself. In this case, the computation of rf relies on a fixpoint computation, which converges in at most quadratic time (the maximal size of rf), described in Algorithm 2. Essentially, we use the axiom ReadFromRGA to populate the read-from relation and then, apply the axiom RetvalRGA iteratively, using the read-from constraints added in previous steps, until the computation converges.

After computing the read-from relation, our algorithm defines the happens-before relation hb as the transitive closure of ro union rf. This is sound because none of the axioms of these datatypes enforce new happens-before constraints, which are not already captured by ro and rf. Then, it checks whether the hb defined this way is acyclic and satisfies the datatype's axioms that constrain hb, i.e., ReadFromMaximal and ReadAllMaximals(when they are present).

Finally, in the case of LWW and RGA, the algorithm computes a (partial) linearization order that satisfies the corresponding Lin_ axioms. Starting from an initial linearization order which is exactly the happens-before, it computes new constraints by instantiating the universally quantified axioms LinLWW and LinRGA. Since these axioms are not "recursive", i.e., they don't enforce linearization order constraints based on other linearization order constraints, a standard instantiation of these axioms is enough to compute a partial linearization order such that any extension to a total order satisfies the datatype's axioms.

Theorem 2.3.1. *Algorithm 1 returns* true *iff the input history is consistent.*

The following holds because Algorithm 1 runs in polynomial time — the rank depends on the number of quantifiers in the datatype's axioms. Indeed, Algorithm 1 represents a least fixpoint computation which converges in at most a quadratic number of iterations (the maximal size of rf).

Corollary 2.3.1. *The admissibility problem is polynomial time for RGA, and for LWW and MVR on differentiated histories.*

Input: History $h = (\mathsf{Op}, \mathsf{ro})$, prefix map m, and set $seen$ of invalid prefix maps
Output: $false$ if there exists no read-from and happens-before relations rf and hb such that $m \subseteq \mathsf{hb}$, and $\langle h, \mathsf{rf}, \mathsf{hb} \rangle$ satisfies the counter axioms.

if m *is complete* **then return** *true*;
foreach *replica* i **do**
  foreach *replica* $j \neq i$ **do**
    $m' \leftarrow m[i \leftarrow m(i) \cup m(j)]$;
    if $m' \notin seen$ *and* $\mathsf{checkCounter}(h, m', seen)$ **then**
      return *true*;
    $seen \leftarrow seen \cup \{m'\}$;
  if $\exists o_1.\ \mathsf{ro}^1(\mathsf{last}_i(m), o_1)$ **then**
    if $\mathsf{meth}(o_1) = \mathsf{read}$ *and* $\mathsf{arg}(o_1) = x \wedge \mathsf{ret}(o_1) \neq |\{o \in m[i] \mid o = \mathsf{inc}(x)\}| - |\{o \in m[i] \mid o = \mathsf{dec}(x)\}|$ **then**
      return *false*;
    $m' \leftarrow m[i \leftarrow m(i) \cup \{o_1\}]$;
    if $m' \notin seen$ *and* $\mathsf{checkCounter}(h, m', seen)$ **then**
      return *true*;
    $seen \leftarrow seen \cup \{m'\}$;
return *false*;

Algorithm 3: The procedure $\mathsf{checkCounter}$, where ro^1 denotes immediate ro-successor, and $f[a \leftarrow b]$ updates function f with mapping $a \mapsto b$.

2.3.2 Replicated Counters

In this section, we propose a sound polynomial time algorithm for replicated counter datatype assuming the number of replicas in the input history is fixed (*i.e.* the width of the rpelica order ro is fixed). The algorithm constructs a valid happens-before order (note that the semantics of the replicated counter doesn't constrain the linearization order) incrementally, following the replica order. At any time, the happens-before order is uniquely determined by a *prefix mapping* that associates to each replica a *prefix* of the history, i.e., a set of operations which is downward-closed w.r.t. replica order (i.e., if it contains an operation it contains all of its ro predecessors). This models the fact that the replica order is included in the happens-before and therefore, if an operation o_1 happens-before another operation o_2, then all the ro predecessors of o_1 happen-before o_2. The happens-before order can be extended in two ways: (1) adding an operation issued on the replica i to the prefix of replica i, or (2) "merging" the prefix of a replica j to the prefix of a replica i (this models the delivery of an operation issued on replica j and all its happens-before predecessors to the replica i). Verifying that an extension of the happens-before is valid, i.e., that the return values of newly-added read operations satisfy the RetvalCounter axiom, doesn't depend on the happens-before order between the operations in the prefix associated to some replica (it is enough to count the inc and dec operations in that prefix). Therefore, the algorithm can be seen as a search in the space of prefix mappings. If the number of replicas in the input history is fixed, then the number of possible prefix mappings is polynomial in the size of the history, which implies that the search can be done in polynomial time.

Let $h = (\mathsf{Op}, \mathsf{ro})$ be a history. To simplify the notations, we assume that the replica order is a union of sequences, each sequence representing the operations issued on the same replica.

Therefore, each operation $o \in \mathsf{Op}$ is associated with a replica identifier $\mathsf{rep}(o) \in [1..n_h]$, where n_h is the number of replicas in h.

A *prefix* of h is a set of operation $\mathsf{Op}' \subseteq \mathsf{Op}$ such that all the ro predecessors of operations in Op' are also in Op', i.e., $\forall o \in \mathsf{Op}.\ \mathsf{ro}^{-1}(o) \in \mathsf{Op}$. Note that the union of two prefixes of h is also a prefix of h. The *last operation* of replica i in a prefix Op' is the ro-maximal operation o with $\mathsf{rep}(o) = i$ included in Op'. A prefix Op' is called *valid* if $(\mathsf{Op}', \mathsf{ro}')$, where ro' is the projection of ro on Op', is admitted by the replicated counter.

A *prefix map* is a mapping m which associates a prefix of h to each replica $i \in [1..n_h]$. Intuitively, a prefix map defines for each replica i the set of operations which are "known" to i, i.e., happen-before the last operation of i in its prefix. Formally, a prefix map m is *included* in a happens-before relation hb, denoted by $m \subseteq \mathsf{hb}$, if for each replica $i \in [1..n_h]$, $\mathsf{hb}(o, o_i)$ for each operation in $o \in m(i) \setminus \{o_i\}$, where o_i is the last operation of i in $m(i)$. We call o_i the *last operation* of i in m, and denoted it by $\mathsf{last}_i(m)$. A prefix map m is *valid* if it associates a valid prefix to each replica, and *complete* if it associates the whole history h to each replica i.

Algorithm 3 lists our algorithm for checking consistency of replicated counter histories. It is defined as a recursive procedure $\mathsf{checkCounter}$ that searches for a sequence of valid extensions of a given prefix map (initially, this prefix map is empty) until it becomes complete. The axiom RetvalCounter is enforced whenever extending the prefix map with a new read operation (when the last operation of a replica i is "advanced" to a read operation). The following theorem states the correctness of the algorithm.

Theorem 2.3.2. $\mathsf{checkCounter}(h, \emptyset, \emptyset)$ *returns* false *if the input history is inconsistent.*

When the number of replicas is fixed, the number of prefix maps becomes polynomial in the size of the history. This follows from the fact that prefixes are uniquely defined by their ro-maximal operations, whose number is fixed. Since the possible number of prefix-map is polynomial when the number of replicas is fixed, the algorithm 3 terminates after exploring polynomially many states. Since the each step of the recursion happens in polynomial time, the algorithm always run in polynomial time in the size of the history when the number of replicas is fixed.

Incompleteness. As a correction of our previous work [16], we show that Algorithm 3 is actually *incomplete*, i.e., it may return false while the history is admissible. The history of a replicated counter in Figure 2.5 is a counterexample to completeness. This history admits a single read-from relation as witness of admissibility, which is given by the edges in the figure. $[\mathsf{inc}(a)]_{r_1}$ must propagate after $[\mathsf{read}(a) = 2]_{r_2}$ and before $[\mathsf{read}(a) = 3]_{r_2}$. To simulate this propagation, the algorithm must reach a prefix map which had $[\mathsf{inc}(a)]_{r_1}$ and $[\mathsf{read}(a) = 2]_{r_2}$ as the ro-maximal operations from each replica. Symmetrically, the same argument holds when $[\mathsf{inc}(a)]_{r_2}$ needs to propagate after $[\mathsf{read}(a) = 2]_{r_1}$ and before $[\mathsf{read}(a) = 3]_{r_1}$. So the algorithm must reach another prefix map which had $[\mathsf{inc}(a)]_{r_2}$ and $[\mathsf{read}(a) = 2]_{r_1}$ as the ro-maximal operations from each replica.

Since the algorithm always extends the maintained prefix map *i.e.* when successful, the sequence of valid extensions from the empty prefix map are always related by inclusion. But these two prefix-maps are *not* related by inclusion. So no sequence of extensions of empty prefix map will see both of them together, and the algorithm will return false because at least one of the $\mathsf{read}(a) = 3$ from one replica will be unsuccessful.

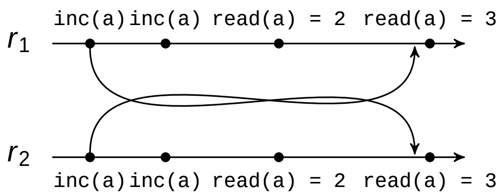


Figure 2.5: An admissible execution of replicated counter for which Algorithm 3 returns false.

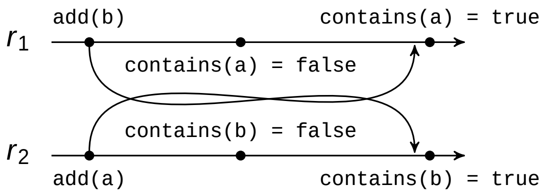


Figure 2.6: An admissible execution of replicated Add-Wins set for which Algorithm 4 returns false.

To construct hb incrementally, we would need to propagate a partial hb at arbitrary future operations at each replica. Naively, this requires maintaining a prefix map at each read operation which is not included in current prefix map. Although the number of possible prefix maps is polynomially bounded for a given history with a bounded number of replicas, maintaining n prefix maps at each read where n is linear in the size of the history, creates exponentially many possible states to explore. The asymptotic complexity of checking admissibility for a Counter history with a bounded number of replicas remains an open question.

2.3.3 Sets and Flags

While Theorem 2.2.1 shows that the admissibility problem is NP-complete for replicated sets and flags even if the number of replicas is fixed, we propose a sound algorithm which runs in polynomial time when additionally, the number of values added to the set, or the number of flags, is also fixed. Note that this doesn't limit the number of operations in the input history which can still be arbitrarily large. In the following, we focus on the Add-Wins Set, the other cases being very similar.

The algorithm for checking consistency is actually an extension of the one presented in Section 2.3.2 for replicated counters. The additional complexity in checking consistency for the Add-Wins Set comes from the validity of $\mathsf{contains}(x)$ return values which requires identifying the maximal predecessors in the happens-before relation that add or remove x (which are not necessarily the maximal hb-predecessors all-together). In the case of counters, it was enough just to count happens-before predecessors. Therefore, we extend the algorithm for replicated counters such that along with the prefix map, we also keep track of the hb-maximal $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ operations for each element x and each replica i. When extending a prefix map with a $\mathsf{contains}$ operation, these hb-maximal operations (which define a witness for the read-from relation) are enough to verify the RetValSet axiom. Extending the prefix of a replica with an add or remove operation (issued on the same replica), or by merging the prefix of another replica, may require an update of these hb-maximal predecessors.

To represent the maximal hb-predecessors, we use a mapping u, called *read-from map*, that associates a set of operations $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ on different replicas to each replica i and element x. Note that two operations on the same replica are necessarily related by hb and cannot be both maximal. A pair of prefix-map m and read-from map u defines a partial read-from relation that associates all the operations in $u(x, i)$ to the last operation of i, i.e., $\mathsf{last}_i(m)$, if this is a $\mathsf{contains}(x)$ operation. For a given read-from relation rf, $\langle m, u\rangle \subseteq \mathsf{rf}$ denotes the fact that this partial read-from relation is included in rf. A prefix $m(i)$ is called valid in the context of a read-from map u if it is admitted by the Add-Wins Set with a read-from relation rf such that $\langle m, u\rangle \subseteq \mathsf{rf}$. A pair $\langle m, u\rangle$ is called valid if $m(i)$ is valid for each replica i.

Algorithm 4 lists our algorithm for checking consistency of Add-Wins Set histories. As for replicated counters, it is defined as a recursive procedure CheckAWSet that searches for a sequence of valid extensions of a given prefix map and read-from map (initially, both of them are empty) until the prefix map becomes complete.

Input: A history $h = (Op, ro)$, a prefix map m, a read-from map u, and a set $seen$ of invalid prefix map and read-from map pairs.
Output: $false$ if there exists no read-from relation rf and happens-before order hb such that $m \subseteq \mathsf{hb}$, $\langle m, u\rangle \subseteq \mathsf{rf}$, and $\langle h, \mathsf{rf}, \mathsf{hb}\rangle$ satisfies the replicated Add-Wins Set axioms.

```
if m is complete then
    return true;
foreach replica i do
    foreach replica j ≠ i do
        m' ← m[i ← m(i) ∪ m(j)];
        u'(x) ← u(x)[i ← (u(x,i) \ (m(j) \ u(x,j))) ∪ (u(x,j) \ (m(i) \ u(x,i)))];
        if ⟨m',u'⟩ ∉ seen and CheckAWSet(h,m',u',seen) then
            return true;
        seen ← seen ∪ {⟨m',u'⟩};
    if ∃o1. ro^1(last_i(m), o1) then
        u' ← u;
        if meth(o1) = contains then
            if ret(o1) = true ⇔ ∄o2 ∈ u(arg(o1), i) s.t. meth(o2) = add ∧ arg(o2) = arg(o1) then
                return false;
        else
            u'(arg(o1), i) ← {o1};
        m' ← m[i ← m(i) ∪ {o1}];
        if ⟨m',u'⟩ ∉ seen and CheckAWSet(h,m',u',seen) then
            return true;
        seen ← seen ∪ {⟨m',u'⟩};
return false;
```

Algorithm 4: The procedure CheckAWSet for checking consistency of Add-Wins Set.

The following theorem states the correctness of the algorithm.

Theorem 2.3.3. *CheckAWSet$(h, \emptyset, \emptyset, \emptyset)$ returns false if the input history is not consistent for the Add-Wins Set.*

When the number of replicas and elements are fixed, the number of read-from maps is polynomial in the size of the history — recall that the number of operations associated by a read-from map to a replica and set element is bounded by the number of replicas. Since the possible number of prefix-map and read-from map is polynomial when the number of replicas and elements are fixed, the algorithm 4 terminates after exploring polynomially many states. Since the each step of the recursion happens in polynomial time, the algorithm always run in polynomial time in the size of the history when the number of replicas and elements are fixed.

Incompleteness. This algorithm can be shown to be incomplete in a way similar to the Counter case. This corrects a statement we have made in our previous work [16]. The history of Add-Wins Set in Figure 2.6 is admissible while Algorithm 4 returns *false*. The explanations are similar to the Counter example in Figure 2.5.

2.4 Related Work

Many have considered consistency models applicable to CRDTs, including causal consistency [68], sequential consistency [67], linearizability [60], session consistency [90], eventual consistency [91], and happens-before consistency [72]. Burckhardt et al. [26, 29] propose a unifying framework to formalize these models. Many have also studied the complexity of verifying data-type agnostic notions of consistency, including serializability, sequential consistency, and linearizability [5, 14, 20, 46, 54, 58, 80], as well as causal consistency [21]. Our definition of the replicated LWW register corresponds to the notion of causal convergence in [21]. This work studies the complexity of the admissibility problem for the replicated LWW register. It shows that this problem is NP-complete in general and polynomial time when each value is written only once. Our NP-completeness result is stronger since it assumes a fixed number of replicas, and our algorithm for the case of unique values is more general and can be applied uniformly to MVR and RGA. While Bouajjani et al. [19, 42] considers the complexity for individual linearizable collection types, we are the first to establish (in)tractability of individual replicated data types. Others have developed effective consistency checking algorithms for sequential consistency [15, 27, 59, 84], serializability [36, 45, 46, 57], linearizability [28, 44, 70, 95], and even weaker notions like eventual consistency [22] and sequential happens-before consistency [41, 43]. In contrast, we are the first to establish precise polynomial-time algorithms for runtime verification of replicated data types.

2.5 Conclusion

In this chapter, we studied various CRDTs, namely Counter, Set, Flag, Registers, Growing Array. We provide novel formal characterizations for these replicated data types, and study the asymptotic complexity of checking conformance for a given execution. We provide polynomial time algorithms for Growing Array and Registers (when the read-from relation is fixed). For other data types, we prove NP-completeness results based on polynomial time reductions from SAT problems. Then, we provide sound polynomial time algorithms for Counters, Set, Flag when the number of replicas and/or elements are bounded by a fixed constant. Since the latter algorithms are not complete, the asymptotic complexity remains open in these cases.

3 Checking Transactional Consistency

In this chapter, we consider the issue of automated testing for transactional databases. More precisely, we focus on the complexity of checking correctness of an execution w.r.t. some transactional consistency model. We consider several consistency models that are the most prevalent in practice: *Read Committed* (RC) [12], *Read Atomic* (RA) [31], *Causal Consistency* (CC) [68], *Prefix Consistency* (PC) [30], *Snapshot Isolation* (SI) [12], and Serializability (SER) [80]. In case of intractability, we introduce algorithms that are polynomial time assuming fixed bounds for certain parameters of the input executions, e.g., the number of sessions.

We define a new specification framework for these consistency models that relies on the fact that the *write-read* relation in an execution (also known as *read-from*), relating reads with the transactions that wrote their value, can be defined effectively. The write-read relation can be extracted easily from executions where each value is written at most once (a variable can be written an arbitrary number of times). This can be easily enforced by tagging values with unique identifiers (e.g., a local counter that is incremented with every new write coupled with a client/session identifier)[1]. Since practical database implementations are data-independent [96], i.e., their behavior doesn't depend on the concrete values read or written in the transactions, any potential buggy behavior can be exposed in executions where each value is written at most once. Therefore, this assumption is without loss of generality.

Previous work [21, 29, 31] has formalized such consistency models using two auxiliary relations: a *visibility* relation defining for each transaction the set of transactions it observes, and a *commit order* defining the order in which transactions are committed to the "global" memory. An execution satisfying some consistency model is defined as the existence of a visibility relation and a commit order obeying certain axioms. In our case, the write-read relation derived from the execution plays the role of the visibility relation. This simplification allows us to state a series of axioms defining these consistency models, which have a common shape. Intuitively, they define lower bounds on the set of transactions t_1 that *must* precede in commit order a transaction t_2 that is read in the execution. Besides shedding a new light on the differences between these consistency models, these axioms are essential for the algorithmic issues we investigate afterwards.

We establish the precise complexity for checking whether an execution satisfies RC, RA, or CC is polynomial time, while the same problem is NP-complete for PC and SI. Moreover, in the case of the NP-complete consistency models (PC, SI, SER), we show that their verification problem becomes polynomial time provided that, roughly speaking, the number of sessions in the input executions is considered to be fixed (i.e., not counted for in the input size). We extend these results even further by relying on an abstraction of executions called *communication graphs* [33]. Roughly

[1]This is also used in Jepsen, e.g., checking dirty reads in Galera [64].

speaking, the vertices of a communication graph correspond to sessions, and the edges represent the fact that two sessions access (read or write) the same variable. We show that all these criteria are polynomial-time checkable provided that the *biconnected* components of the communication graph are of fixed size.

We provide an experimental evaluation of our algorithms on executions of CockroachDB [34], which claims to implement serializability [35] acknowledging however the possibility of anomalies, Galera [49], whose documentation contains contradicting claims about whether it implements snapshot isolation [50, 51], and AntidoteDB [6], which claims to implement causal consistency [7]. Our implementation reports violations of these criteria in all cases. The consistency violations we found for AntidoteDB are novel and have been confirmed by its developers. We show that our algorithms are efficient and scalable. In particular, we show that, although the asymptotic complexity of our algorithms is exponential in general (w.r.t. the number of sessions), the worst-case behavior is not exercised in practice.

The remainder of this chapter is organized as follows:

- Section 3.1 defines a new specification framework for describing common transactional-consistency criteria;
- Section 3.2 shows that checking RC, RA, and CC is polynomial time while checking PC and SI is NP-complete;
- Section 3.3 and Section 3.4 show that PC, SI, and SER are polynomial-time checkable assuming that the communication graph of the input execution has fixed-size biconnected components;
- Section 3.5 describes an empirical evaluation of our algorithms on executions generated by production databases;

Section 3.6 overviews related work, and Section 3.7 concludes.

3.1 Consistency Criteria

3.1.1 Histories

We consider a transactional database storing a set of variables $\mathsf{Var} = \{x, y, \ldots\}$. Clients interact with the database by issuing transactions formed of read and write operations. Assuming an unspecified set of values Val and a set of operation identifiers OId, we let

$$\mathsf{Op} = \{\mathsf{read}_i(x, v), \mathsf{write}_i(x, v) : i \in \mathsf{OId}, x \in \mathsf{Var}, v \in \mathsf{Val}\}$$

be the set of operations reading a value v or writing a value v to a variable x. We omit operation identifiers when they are not important.

Definition 3.1.1. *A* transaction log $\langle t, O, \mathsf{po}\rangle$ *is a transaction identifier* t *and a finite set of operations* O *along with a strict total order* po *on* O*, called* program order.

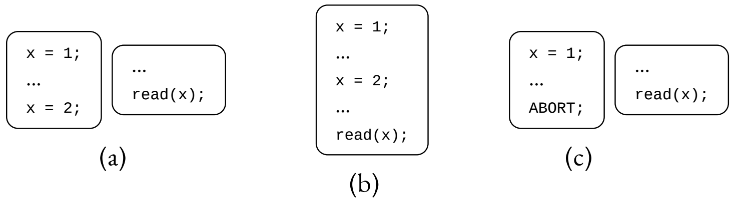


Figure 3.1: Examples of transactions used to justify our simplifying assumptions (each box represents a different transaction): (a) only the last written value is observable in other transactions, (b) reads following writes to the same variable return the last written value in the same transaction, and (c) values written in aborted transactions are not observable.

The program order po represents the order between instructions in the body of a transaction. We assume that each transaction log is well-formed in the sense that if a read of a variable x is preceded by a write to x in po, then it should return the value written by the last write to x before the read (w.r.t. po). This property is implicit in the definition of every isolation level or consistency model that we are aware of. For simplicity, we may use the term *transaction* instead of transaction log and ignore transaction identifier assuming all transaction is uniquely identified. The set of all transaction logs is denoted by Tlogs.

We use t, t_1, t_2, $\ldots$ to range over transactions. The set of read, resp., write, operations in a transaction t is denoted by $\mathsf{reads}(t)$, resp., $\mathsf{writes}(t)$. The extension to sets of transactions is defined as usual. Also, we say that a transaction t *writes* a variable x, denoted by t writes x, when $\mathsf{write}_i(x, v) \in \mathsf{writes}(t)$ for some i and v. Similarly, a transaction t *reads* a variable x when $\mathsf{read}_i(x, v) \in \mathsf{reads}(t)$ for some i and v.

To simplify the exposition, we assume that each transaction t contains at most one write operation to each variable[2], and that a read of a variable x cannot be preceded by a write to x in the same transaction[3]. If a transaction would contain multiple writes to the same variable, then only the last one should be visible to other transactions (w.r.t. any consistency criterion considered in practice). For instance, the `read(x)` in Figure 3.1a should not return 1 because this is not the last value written to `x` by the other transaction. It can return the initial value or 2. Also, if a read would be preceded by a write to the same variable in the same transaction, then it should return a value written in the same transaction (i.e., the value written by the latest write to x in that transaction). For instance, the `read(x)` in Figure 3.1b can only return 2 (assuming that there are no other writes on `x` in the same transaction). These two properties can be verified easily (in a syntactic manner) on a given execution. Beyond these two properties, the various consistency criteria used in practice constrain only the last writes to each variable in each transaction and the reads that are not preceded by writes to the same variable in the same transaction.

Consistency criteria are formalized on an abstract view of an execution called *history*. A history includes only successful or committed transactions. In the context of databases, it is always assumed that the effect of aborted transactions should not be visible to other transactions, and therefore, they can be ignored. For instance, the `read(x)` in Figure 3.1c should not return the value 1 written by the aborted transaction. The transactions are ordered according to a (partial) *session*

[2]That is, for every transaction t, and every $\mathsf{write}(x, v), \mathsf{write}(y, v') \in \mathsf{writes}(t)$, we have that $x \neq y$.

[3]That is, for every transaction $t = \langle O, \mathsf{po} \rangle$, if $\mathsf{write}(x, v) \in \mathsf{writes}(t)$ and there exists $\mathsf{read}(x, v) \in \mathsf{reads}(t)$, then we have that $\langle \mathsf{read}(x, v), \mathsf{write}(x, v) \rangle \in \mathsf{po}$

order so which represents ordering constraints imposed by the applications using the database. Most often, so is a union of sequences, each sequence being called a *session*. We assume that the history includes a *write-read* relation that identifies the transaction writing the value returned by each read in the execution. As mentioned before, such a relation can be extracted easily from executions where each value is written at most once. Since in practice, databases are data-independent [96], i.e., their behavior does not depend on the concrete values read or written in the transactions, any potential buggy behavior can be exposed in such executions.

Definition 3.1.2. *A* history $\langle T, \mathsf{so}, \mathsf{wr} \rangle$ *is a set of transactions* T *along with a strict partial order* so *called* session order, *and a relation* $\mathsf{wr} \subseteq T \times \mathsf{reads}(T)$ *called* write-read *relation, s.t.*

- *the inverse of* wr *is a total function, and if* $(t, \mathsf{read}(x, v)) \in \mathsf{wr}$, *then* $\mathsf{write}(x, v) \in t$, *and*
- $\mathsf{so} \cup \mathsf{wr}$ *is acyclic.*

To simplify the technical exposition, we assume that every history includes a distinguished transaction writing the initial values of all variables. This transaction precedes all the other transactions in so. We use $h, h_1, h_2, \ldots$ to range over histories.

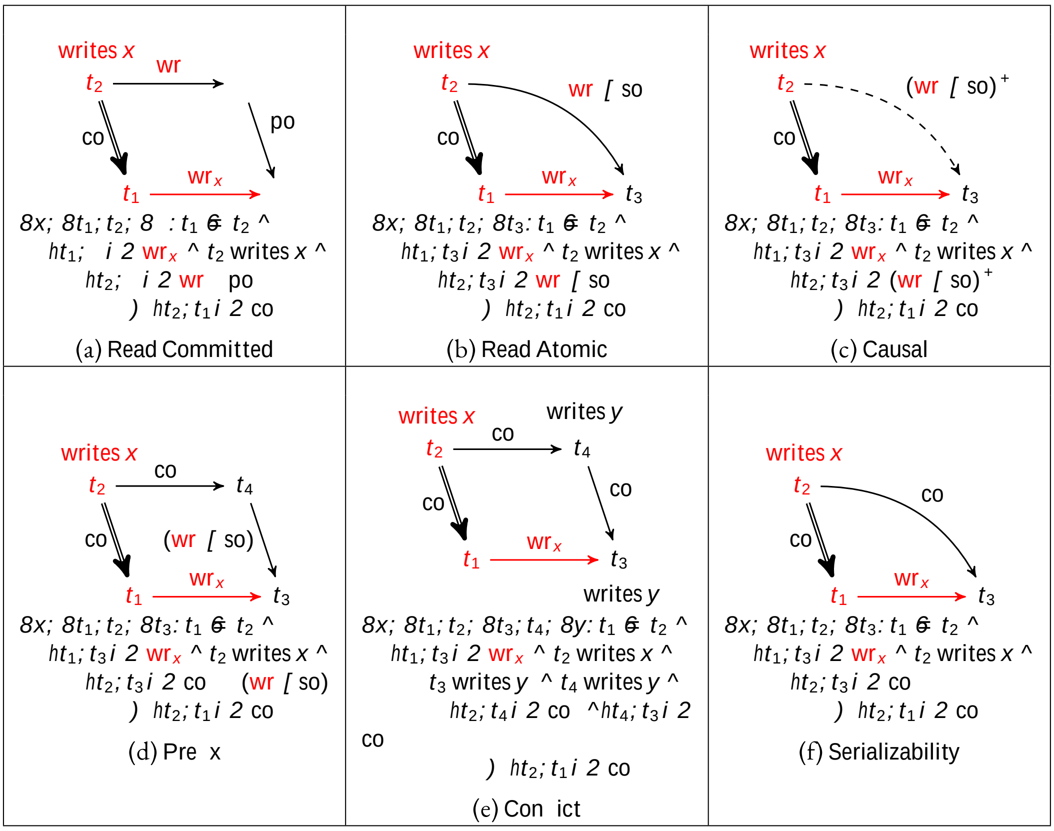
We say that the read operation $\mathsf{read}(x, v)$ reads value v from variable x written by t when $(t, \mathsf{read}(x, v)) \in \mathsf{wr}$. For a given variable x, wr_x denotes the restriction of wr to reads of variable x, *i.e.* , $\mathsf{wr}_x = \mathsf{wr} \cap (T \times \{\mathsf{read}(x, v) \mid v \in \mathsf{Val}\})$. Moreover, we extend the relations wr and wr_x to pairs of transactions as follows: $\langle t_1, t_2 \rangle \in \mathsf{wr}$, resp., $\langle t_1, t_2 \rangle \in \mathsf{wr}_x$, iff there exists a read operation $\mathsf{read}(x, v) \in \mathsf{reads}(t_2)$ such that $\langle t_1, \mathsf{read}(x, v) \rangle \in \mathsf{wr}$, resp., $\langle t_1, \mathsf{read}(x, v) \rangle \in \mathsf{wr}_x$. We say that the transaction t_1 is *read* by the transaction t_2 when $\langle t_1, t_2 \rangle \in \mathsf{wr}$, and that it is *read* when it is read by some transaction t_2.

3.1.2 Axiomatic Framework

We describe an axiomatic framework to characterize the set of histories satisfying a certain consistency criterion. The overarching principle is to say that a history satisfies a certain criterion if there exists a strict total order on its transactions, called *commit order* and denoted by co, which extends the write-read relation and the session order, and which satisfies certain properties. These properties are expressed by a set of axioms that relate the commit order with the session-order and the write-read relation in the history.

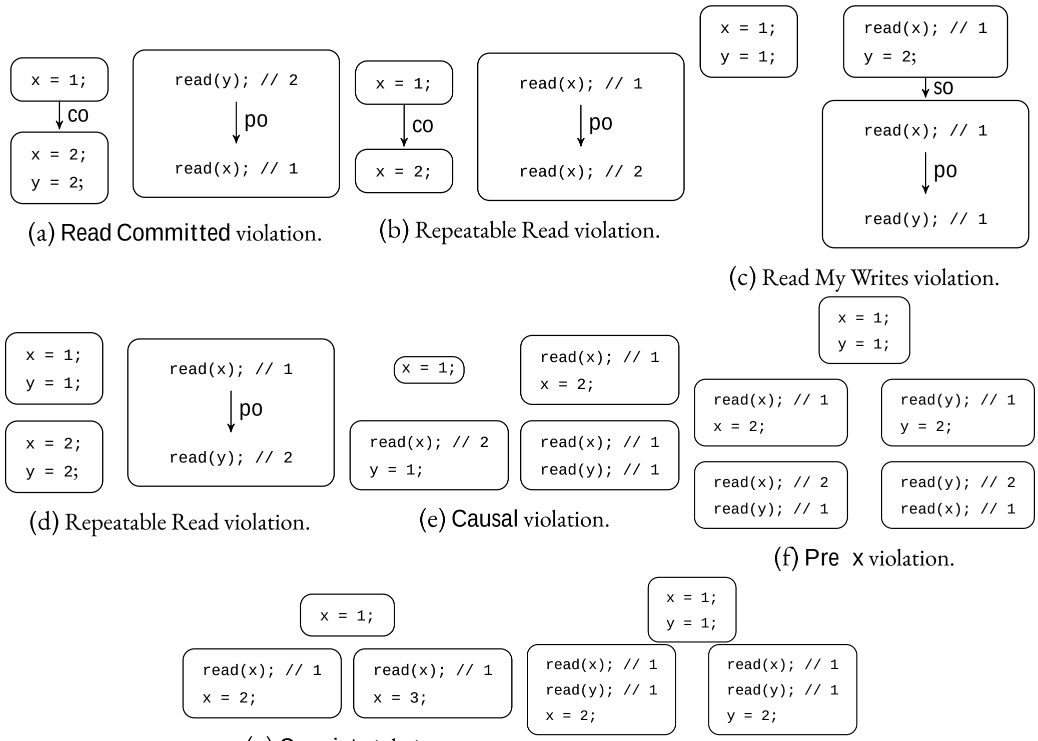
The axioms we use have a uniform shape: they define mandatory co predecessors t_2 of a transaction t_1 that is read in the history. For instance, the criterion called Read Committed (RC) [12] requires that every value read in the history was written by a committed transaction, and also, that the reads in the same transaction are "monotonic" in the sense that they do not return values that are older, w.r.t. the commit order, than other values read in the past[4]. While the first condition holds for any history (because of the surjectivity of wr), the second condition is expressed by the axiom $\mathsf{Read\ Committed}$ in Figure 3.2a. This axiom states that for any transaction t_1 writing a variable x that is read in a transaction t, the set of transactions t_2 writing x and read previously in the same transaction must precede t_1 in commit order. For instance, Figure 3.3a shows a history

[4] This monotonicity property corresponds to the fact that in the original formulation of Read Committed [12], every write is guarded by the acquisition of a lock on the written variable, that is held until the end of the transaction.



(a) Read Committed (b) Read Atomic (c) Causal (d) Prefix (e) Conflict (f) Serializability

Figure 3.2: Definitions of consistency axioms. The reflexive and transitive, resp., transitive, closure of a relation rel is denoted by rel^*, resp., rel^+. Also, $;$ denotes the composition of two relations, i.e., $rel_1 ; rel_2 = \{\langle a, b \rangle \mid \exists c. \langle a, c \rangle \in rel_1 \wedge \langle c, b \rangle \in rel_2\}$.



(a) Read Committed violation.

(b) Repeatable Read violation.

(c) Read My Writes violation.

(d) Repeatable Read violation.

(e) Causal violation.

(f) Prefix violation.

(g) Conflict violation.

(h) Serializability violation.

Figure 3.3: Examples of histories used to explain the axioms in Figure 3.2. For readability, the wr relation is defined by the values written in comments with each read.

and a (partial) commit order that does not satisfy this axiom because `read(x)` returns the value written in a transaction "older" than the transaction read in the previous `read(y)`. An example of a history and commit order satisfying this axiom is given in Figure 3.3b.

More precisely, the axioms are first-order formulas[5] of the following form:

$$\forall x, \forall t_1, t_2, \forall \alpha.\ t_1 \neq t_2 \wedge \langle t_1, \alpha \rangle \in \mathsf{wr}_x \wedge t_2 \text{ writes } x \wedge \phi(t_2, \alpha) \Rightarrow \langle t_2, t_1 \rangle \in \mathsf{co} \tag{3.1}$$

where ϕ is a property relating t_2 and α (i.e., the read or the transaction reading from t_1) that varies from one axiom to another. Intuitively, this axiom schema states the following: in order for α to read specifically t_1's write on x, it must be the case that every t_2 that also writes x and satisfies $\phi(t_2, \alpha)$ was committed before t_1. Note that in all cases we consider, $\phi(t_2, \alpha)$ already ensures that t_2 is committed before the read α, so this axiom schema ensures that t_2 is furthermore committed before t_1's write.

The axioms used throughout the chapter are given in Figure 3.2. The property ϕ relates t_2 and α using the write-read relation and the session order in the history, and the commit order.

In the following, we explain the rest of the consistency criteria we consider and the axioms defining them. Read Atomic (RA) [31] is a strengthening of Read Committed defined by the axiom Read Atomic, which states that for any transaction t_1 writing a variable x that is read in a transaction t_3, the set of wr or so predecessors of t_3 writing x must precede t_1 in commit order. The case of wr predecessors corresponds to the Repeatable Read criterion in [12], which requires that successive reads of the same variable in the same transaction return the same value, Figure 3.3b showing a violation, and also that every read of a variable x in a transaction t returns the value written by the maximal transaction t' (w.r.t. the commit order) that is read by t, Figure 3.3d showing a violation (for any commit order between the transactions on the left, either `read(x)` or `read(y)` will return a value not written by the maximal transaction). The case of so predecessors corresponds to the "read-my-writes" guarantee [90] concerning sessions, which states that a transaction t must observe previous writes in the same session. For instance, `read(y)` returning 1 in Figure 3.3c shows that the last transaction on the right does not satisfy this guarantee: the transaction writing 1 to y was already visible to that session before it wrote 2 to y, and therefore the value 2 should have been read. Read Atomic requires that the so predecessor of the transaction reading y be ordered in co before the transaction writing 1 to y, which makes the union $\mathsf{co} \cup \mathsf{wr}$ cyclic.

The following lemma shows that for histories satisfying Read Atomic, the inverse of wr_x extended to transactions is a total function.

Lemma 3.1.1. *Let $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$ be a history. If $\langle h, \mathsf{co} \rangle$ satisfies* Read Atomic, *then for every transaction t and two reads $\mathsf{read}_{i_1}(x, v_1), \mathsf{read}_{i_2}(x, v_2) \in \mathsf{reads}(t)$, $\mathsf{wr}^{-1}(\mathsf{read}_{i_1}(x, v_1)) = \mathsf{wr}^{-1}(\mathsf{read}_{i_2}(x, v_2))$ and $v_1 = v_2$.*

Proof. Let $\langle t_1, \mathsf{read}_{i_1}(x, v_1) \rangle, \langle t_2, \mathsf{read}_{i_2}(x, v_2) \rangle \in \mathsf{wr}_x$. Then t_1, t_2 write to x. Let us assume by contradiction, that $t_1 \neq t_2$. By Read Atomic, $\langle t_2, t_1 \rangle \in \mathsf{co}$ because $\langle t_1, \mathsf{read}_{i_1}(x, v_1) \rangle \in \mathsf{wr}_x$ and t_2 writes to x. Similarly, we can also show that $\langle t_1, t_2 \rangle \in \mathsf{co}$. This contradicts the

[5] These formulas are interpreted on tuples $\langle h, \mathsf{co} \rangle$ of a history h and a commit order co on the transactions in h as usual.

Table 3.1: Consistency model definitions

Consistency model	Axioms
READ COMMITTED (RC)	Read Committed
READ ATOMIC (RA)	Read Atomic
CAUSAL CONSISTENCY (CC)	Causal
PREFIX CONSISTENCY (PC)	Prefix
SNAPSHOT ISOLATION (SI)	Prefix $\wedge$ Conflict
SERIALIZABILITY (SER)	Serializability

fact that co is a strict total order. Therefore, $t_1 = t_2$. We also have that $v_1 = v_2$ because each transaction contains a single write to x. □

CAUSAL CONSISTENCY (CC) [68] is defined by the axiom Causal, which states that for any transaction t_1 writing a variable x that is read in a transaction t_3, the set of $(\mathsf{wr} \cup \mathsf{so})^+$ predecessors of t_3 writing x must precede t_1 in commit order ($(\mathsf{wr} \cup \mathsf{so})^+$ is usually called the *causal* order). A violation of this axiom can be found in Figure 3.3e: the transaction t_2 writing 2 to x is a $(\mathsf{wr} \cup \mathsf{so})^+$ predecessor of the transaction t_3 reading 1 from x because the transaction t_4, writing 1 to y, reads x from t_2 and t_3 reads y from t_4. This implies that t_2 should precede in commit order the transaction t_1 writing 1 to x, which again, is inconsistent with the write-read relation (t_2 reads from t_1).

PREFIX CONSISTENCY (PC) [30] is a strengthening of CC, which requires that every transaction observes a prefix of a commit order between all the transactions. With the intuition that the observed transactions are $\mathsf{wr} \cup \mathsf{so}$ predecessors, the axiom Prefix defining PC, states that for any transaction t_1 writing a variable x that is read in a transaction t_3, the set of co predecessors of transactions observed by t_3 writing x must precede t_1 in commit order (we use co to say that even the transactions observed by t_3 must precede t_1). This ensures the prefix property stated above. An example of a PC violation can be found in Figure 3.3f: the two transactions on the bottom read from the three transactions on the top, but any serialization of those three transactions will imply that one of the combinations x=1, y=2 or x=2, y=1 cannot be produced at the end of a prefix in this serialization.

SNAPSHOT ISOLATION (SI) [12] is a strengthening of PC that disallows two transactions to observe the same prefix of a commit order if they *conflict*, i.e., write to a common variable. It is defined by the conjunction of Prefix and another axiom called Conflict, which requires that for any transaction t_1 writing a variable x that is read in a transaction t_3, the set of co predecessors writing x of transactions conflicting with t_3 and before t_3 in commit order, must precede t_1 in commit order. Figure 3.3g shows a Conflict violation.

Finally, SERIALIZABILITY (SER) [80] is defined by the axiom with the same name, which requires that for any transaction t_1 writing to a variable x that is read in a transaction t_3, the set of co predecessors of t_3 writing x must precede t_1 in commit order. This ensures that each transaction observes the effects of all the co predecessors. Figure 3.3h shows a Serializability violation.

The next lemma states the relationship between these axioms.

Lemma 3.1.2. *The following entailments hold:*

$$\mathsf{Causal} \Rightarrow \mathsf{Read\ Atomic} \Rightarrow \mathsf{Read\ Committed}$$
$$\mathsf{Prefix} \Rightarrow \mathsf{Causal}$$
$$\mathsf{Serializability} \Rightarrow \mathsf{Prefix} \wedge \mathsf{Conflict}$$

Proof. We will show the contrapositive of each implication:

- If $\langle h, \mathsf{co}\rangle$ does not satisfy Read Committed, then

  $$\exists x, \exists t_1, t_2, \exists \alpha, \beta.\ \langle t_1, \alpha\rangle \in \mathsf{wr}_x \wedge t_2 \text{ writes } x \wedge \langle t_2, \beta\rangle \in \mathsf{wr} \wedge \langle \beta, \alpha\rangle \in \mathsf{po} \wedge \langle t_1, t_2\rangle \in \mathsf{co}.$$

  Let t_3 the transaction containing α and β. We have that $\langle t_2, t_3\rangle \in \mathsf{wr}$. But then we have t_1, t_2, t_3 such that $\langle t_1, t_3\rangle \in \mathsf{wr}_x$ and $\langle t_2, t_3\rangle \in \mathsf{wr}$ and t_2 writes x. So by Read Atomic, $\langle t_2, t_1\rangle \in \mathsf{co}$. This contradicts the fact that co is a strict total order. Therefore, $\langle h, \mathsf{co}\rangle$ does not satisfy Read Atomic.

- If $\langle h, \mathsf{co}\rangle$ does not satisfy Read Atomic, then

  $$\exists x, \exists t_1, t_2, t_3.\ \langle t_1, t_3\rangle \in \mathsf{wr}_x \wedge t_2 \text{ writes } x \wedge \langle t_2, t_3\rangle \in \mathsf{wr} \cup \mathsf{so} \wedge \langle t_1, t_2\rangle \in \mathsf{co}.$$

  Then $\langle t_2, t_3\rangle \in (\mathsf{wr} \cup \mathsf{so})^+$. Then, by Causal, we have $\langle t_2, t_1\rangle \in \mathsf{co}$, which contradicts the fact that co is a strict total order. Therefore, $\langle h, \mathsf{co}\rangle$ does not satisfy Causal.

- If $\langle h, \mathsf{co}\rangle$ does not satisfy Causal, then

  $$\exists x, \exists t_1, t_2, t_3.\ \langle t_1, t_3\rangle \in \mathsf{wr}_x \wedge t_2 \text{ writes } x \wedge \langle t_2, t_3\rangle \in (\mathsf{wr} \cup \mathsf{so})^+ \wedge \langle t_1, t_2\rangle \in \mathsf{co}.$$

  But, $(\mathsf{wr} \cup \mathsf{so})^+ = (\mathsf{wr} \cup \mathsf{so})^* \circ (\mathsf{wr} \cup \mathsf{so}) \subseteq \mathsf{co}^* \circ (\mathsf{wr} \cup \mathsf{so})$. Therefore, $\langle t_2, t_3\rangle \in \mathsf{co}^* \circ (\mathsf{wr} \cup \mathsf{so})$. Then, by Prefix, we have $\langle t_2, t_1\rangle \in \mathsf{co}$, which contradicts the fact that co is a strict total order. Therefore, $\langle h, \mathsf{co}\rangle$ does not satisfy Prefix.

- If $\langle h, \mathsf{co}\rangle$ does not satisfy Prefix or Conflict, then

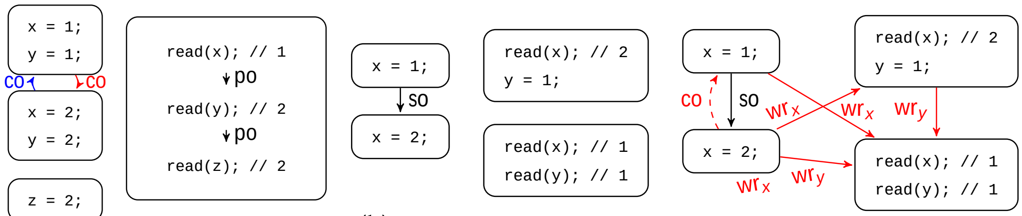
  $$\exists x, \exists t_1, t_2, t_3, t_4.\ \langle t_1, t_3\rangle \in \mathsf{wr}_x \wedge t_2 \text{ writes } x \wedge \langle t_2, t_4\rangle \in \mathsf{co}^* \wedge \langle t_1, t_2\rangle \in \mathsf{co}$$

  and

  - $\langle t_4, t_3\rangle \in \mathsf{co} \wedge t_3$ writes $y \wedge t_3$ writes y if it violates Conflict.
  - $\langle t_4, t_3\rangle \in (\mathsf{wr} \cup \mathsf{so})$ if it violates Prefix.

  In both cases, we have that $\langle t_4, t_3\rangle \in \mathsf{co}$. Because co is transitive, $\langle t_2, t_4\rangle \in \mathsf{co}^*$ and $\langle t_4, t_3\rangle \in \mathsf{co}$ imply that $\langle t_2, t_3\rangle \in \mathsf{co}$. Then by Serializability, we have $\langle t_2, t_1\rangle \in \mathsf{co}$, which contradicts the fact that co is a strict total order. Therefore, $\langle h, \mathsf{co}\rangle$ does not satisfy Serializability.

□



(a) Violation of Read Atomic

(b) Valid w.r.t. Read Atomic

(c) Violation of Causal Consistency

Figure 3.4: Applying the RA and CC checking algorithms.

Definition 3.1.3. *Given a set of axioms X defining a criterion C like in Table 3.1, a history $h = \langle T, \mathsf{so}, \mathsf{wr}\rangle$ satisfies C iff there exists a strict total order co such that $\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{co}$ and $\langle h, \mathsf{co}\rangle$ satisfies X.*

Definition 3.1.3 and Lemma 3.1.2 imply that each consistency criterion in Table 3.1 is stronger than its predecessors (reading them from top to bottom), e.g., CC is stronger than RA and RC. This relation is known to be strict [31], e.g., RA is not stronger than CC.

3.2 Checking Consistency Criteria

This section establishes the complexity of checking the different consistency criteria in Table 3.1 for a given history. More precisely, we show that Read Committed, Read Atomic, and Causal Consistency can be checked in polynomial time while the problem of checking the rest of the criteria is NP-complete.

Intuitively, the polynomial time results are based on the fact that the axioms defining those consistency criteria do not contain the commit order (co) on the left-hand side of the entailment. Therefore, proving the existence of a commit order satisfying those axioms can be done using a saturation procedure that builds a "partial" commit order based on instantiating the axioms on the write-read relation and the session order in the given history. Since the commit order must be an extension of the write-read relation and the session order, it contains those two relations from the beginning. This saturation procedure stops when the order constraints derived this way become cyclic. For instance, let us consider applying such a procedure corresponding to RA on the histories in Figure 3.4a and Figure 3.4b. Applying the axiom in Figure 3.2b on the first history, since the transaction on the right reads 2 from y, we get that its wr_x predecessor (i.e., the first transaction on the left) must precede the transaction writing 2 to y in commit order (the red edge). This holds because the wr_x predecessor writes on y. Similarly, since the same transaction reads 1 from x, we get that its wr_y predecessor must precede the transaction writing 1 to x in commit order (the blue edge). This already implies a cyclic commit order, and therefore, this history does not satisfy RA. On the other hand, for the history in Figure 3.4b, all the axiom instantiations are vacuous, i.e., the left part of the entailment is false, and therefore, it satisfies RA. Checking CC on the history in Figure 3.4c requires a single saturation step: since the transaction on the bottom right reads 1 from x, its $\mathsf{wr}_x \circ \mathsf{wr}_y$ predecessor that writes on x (the transaction on the bottom left) must precede in commit order the transaction writing 1 to x. Since this is already inconsistent with the session order, we get that this history violates CC.

Input: A history $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$
Output: $true$ iff h satisfies CAUSAL CONSISTENCY

1 **if** $\mathsf{so} \cup \mathsf{wr}$ *is cyclic* **then**
2   **return** *false*;
3 $\mathsf{co} \leftarrow \mathsf{so} \cup \mathsf{wr}$;
4 **foreach** $x \in \mathsf{vars}(h)$ **do**
5   **foreach** $t_1 \neq t_2 \in T$ *s.t.* t_1 *and* t_2 *write* x **do**
6   **if** $\exists t_3. \langle t_1, t_3 \rangle \in \mathsf{wr}_x \wedge \langle t_2, t_3 \rangle \in (\mathsf{so} \cup \mathsf{wr})^+$ **then**
7             $\mathsf{co} \leftarrow \mathsf{co} \cup \{\langle t_2, t_1 \rangle\}$;
8 **if** co *is cyclic* **then**
9   **return** *false*;
10 **else**
11   **return** *true*;

Algorithm 5: Checking CAUSAL CONSISTENCY.

Algorithm 5 lists our procedure for checking CC. As explained above, co is initially set to $\mathsf{so} \cup \mathsf{wr}$, and then, it is saturated with other ordering constraints implied by non-vacuous instantiations of the axiom Causal (where the left-hand side of the implication evaluates to true). The algorithms concerning RC and RA are defined in a similar way by essentially changing the test at line 6 so that it corresponds to the left-hand side of the implication in the corresponding axiom. Algorithm 5 can be rewritten as a Datalog program containing straightforward Datalog rules for computing transitive closures and relation composition, and a rule of the form[6]

$$\langle t_2, t_1 \rangle \in \mathsf{co} \text{ :- } t_1 \neq t_2, \langle t_1, t_3 \rangle \in \mathsf{wr}_x, \langle t_2, t_3 \rangle \in (\mathsf{so} \cup \mathsf{wr})^+$$

to represent the Causal axiom. The following is a consequence of the fact that these algorithms run in polynomial time (or equivalently, the corresponding Datalog programs can be evaluated in polynomial time over a database that contains the wr and so relations in a given history).

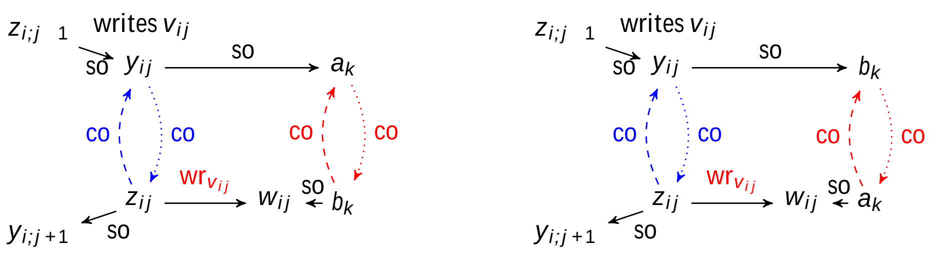
Theorem 3.2.1. *For any criterion $C \in \{$*READ COMMITTED, READ ATOMIC, CAUSAL CONSISTENCY*$\}$, the problem of checking whether a given history satisfies C is polynomial time.*

On the other hand, checking PC, SI, and SER is NP-complete in general. We show this using a reduction from boolean satisfiability (SAT) that covers uniformly all the three cases. In the case of SER, it provides a new proof of the NP-completeness result by [80], which uses a reduction from the so-called *non-circular* SAT and which cannot be extended to PC and SI.

Theorem 3.2.2. *For any criterion $C \in \{$*PREFIX CONSISTENCY, SNAPSHOT ISOLATION, SERIALIZABILITY*$\}$ the problem of checking whether a given history satisfies C is NP-complete.*

Proof. Given a history, any of these three criteria can be checked by guessing a total commit order on its transactions and verifying whether it satisfies the corresponding axioms. This shows that the problem is in NP.

[6] We write Datalog rules using a standard notation $head$:- $body$ where $head$ is a relational atom (written as $\langle a, b \rangle \in R$ where a, b are elements and R a binary relation) and $body$ is a list of relational atoms.



(a) $\ell_{ij} = x_k$ (b) $\ell_{ij} = \neg x_k$

Figure 3.5: Sub-histories included in h_φ for each literal ℓ_{ij} and variable x_k.

To show NP-hardness, we define a reduction from boolean satisfiability. Therefore, let $\varphi = D_1 \wedge \ldots \wedge D_m$ be a CNF formula over the boolean variables $x_1, \ldots, x_n$ where each D_i is a disjunctive clause with m_i literals. Let ℓ_{ij} denote the j-th literal of D_i.

We construct a history h_φ such that φ is satisfiable if and only if h_φ satisfies PC, SI, or SER. Since SER $\Rightarrow$ SI $\Rightarrow$ PC, we show that (1) if h_φ satisfies PC, then φ is satisfiable, and (2) if φ is satisfiable, then h_φ satisfies SER.

Construction of h_φ The main idea of the construction is to represent truth values of each of the variables and literals in φ with the polarity of the commit order between corresponding transaction pairs. For each variable x_k, h_φ contains a pair of transactions a_k and b_k, and for each literal ℓ_{ij}, h_φ contains a set of transactions w_{ij}, y_{ij} and z_{ij}[7]. We want to have that x_k is false if and only if $\langle a_k, b_k\rangle \in \mathsf{co}$, and ℓ_{ij} is false if and only if $\langle y_{ij}, z_{ij}\rangle \in \mathsf{co}$ (the transaction w_{ij} is used to "synchronize" the truth value of the literals with that of the variables, which is explained later).

The history h_φ should ensure that the co ordering constraints corresponding to an assignment that falsifies the formula (*i.e.* one of its clauses) form a cycle. To achieve that, we add all pairs $z_{ij}, y_{i,(j+1)\%m_i}$ in the session order so. An unsatisfied clause D_i, *i.e.* every ℓ_{ij} is false, leads to a cycle of the form $y_{i1} \xrightarrow{\mathsf{co}} z_{i1} \xrightarrow{\mathsf{so}} y_{i2} \xrightarrow{\mathsf{co}} z_{i2} \cdots z_{im_i} \xrightarrow{\mathsf{so}} y_{i1}$.

The most complicated part of the construction is to ensure some consistency between the truth value of the literals and the truth value of the variables, e.g., $\ell_{ij} = x_k$ is true iff x_k is true, for at least one literal ℓ_{ij} interpreted as true in every clause D_i (if such a literal exists). Figure 3.5a shows the sub-history associated to a positive literal $\ell_{ij} = x_k$ while Figure 3.5b shows the case of a negative literal $\ell_{ij} = \neg x_k$. For a positive literal $\ell_{ij} = x_k$ (Figure 3.5a), (1) we enrich session order with the pairs $\langle y_{ij}, a_k\rangle$ and $\langle b_k, w_{ij}\rangle$, (2) we include writes to a variable v_{ij} in the transactions y_{ij} and z_{ij}, and (3) we make w_{ij} read from z_{ij}, *i.e.* $\langle z_{ij}, w_{ij}\rangle \in \mathsf{wr}_{v_{ij}}$. The case of a negative literal is similar, switching the roles of a_k and b_k.

PC for h_φ implies satisfiability of φ If h_φ satisfies PC, then there exists a total commit order co between the transactions described above, which together with h_φ satisfies Prefix. We show that the assignment of the variables x_k explained above (defined by the co order between a_k and b_k, for each k) satisfies the formula φ. For each clause D_i, the so constraints between the

[7] We assume that the transactions a_k and b_k associated to a variable x_k are distinct and different from the transactions associated to another variable $x_{k'} \neq x_k$ or to a literal ℓ_{ij}. Similarly, for the transactions w_{ij}, y_{ij} and z_{ij} associated to a literal ℓ_{ij}.

transactions y_{ij}, z_{ij} with $1 \leq j \leq m_i$ imply that there exist some z_{ij} that is committed before its corresponding y_{ij}. These two transactions are included in the sub-history corresponding to the literal ℓ_{ij} (Figure 3.5a or Figure 3.5b depending on the polarity of the literal).

The definition of this sub-history ensures that the interpretation to true of the literal ℓ_{ij} (given by the order in co between z_{ij} and y_{ij}) is consistent with the assignment of the variable it contains (defined by the co order between a_k, b_k). More precisely, it ensures that if the co goes upwards on the left-hand side ($\langle z_{ij}, y_{ij}\rangle \in \mathsf{co}$) like in this case, then it must also go upwards on the right-hand side ($\langle b_k, a_k\rangle \in \mathsf{co}$ in the case of a positive literal, and $\langle a_k, b_k\rangle \in \mathsf{co}$ in the case of a negative literal) to satisfy Prefix. For instance, if $\ell_{ij} = x_k$ is a positive literal and we assume by contradiction that $\langle a_k, b_k\rangle \in \mathsf{co}$, then $\langle y_{ij}, w_{ij}\rangle \in \mathsf{so} \circ \mathsf{co} \circ \mathsf{so}$. Therefore, for every commit order co such that $\langle h_\varphi, \mathsf{co}\rangle$ satisfies Prefix, $\langle a_k, b_k\rangle \in \mathsf{co}$ implies $\langle y_{ij}, z_{ij}\rangle \in \mathsf{co}$, which contradicts the hypothesis. Indeed, if $\langle a_k, b_k\rangle \in \mathsf{co}$, instantiating the Prefix axiom where y_{ij} plays the role of t_2, z_{ij} plays the role of t_1, and w_{ij} plays the role of t_3, we obtain that $\langle y_{ij}, z_{ij}\rangle \in \mathsf{co}$.

Therefore, the assignment of the variables x_k leads to at least one literal interpreted to true in each clause D_i, and the formula φ is satisfiable.

Satisfiability of φ implies SER for h_φ Let α be a satisfying assignment for φ. Also, let co' be a binary relation that includes so and wr such that if $\alpha(x_k) = \mathit{false}$, then $\langle a_k, b_k\rangle \in \mathsf{co}'$, $\langle y_{ij}, z_{ij}\rangle \in \mathsf{co}'$ for each $\ell_{ij} = x_k$, and $\langle z_{ij}, y_{ij}\rangle \in \mathsf{co}'$ for each $\ell_{ij} = \neg x_k$, and if $\alpha(x_k) = \mathit{true}$, then $\langle b_k, a_k\rangle \in \mathsf{co}'$, $\langle z_{ij}, y_{ij}\rangle \in \mathsf{co}'$ for each $\ell_{ij} = x_k$, and $\langle y_{ij}, z_{ij}\rangle \in \mathsf{co}'$ for each $\ell_{ij} = \neg x_k$. Looking at the sub-histories corresponding to literals ℓ_{ij} (Figure 3.5a or Figure 3.5b), co' goes in the same direction (upwards or downwards) on both sides.

Note that co' is acyclic: no cycle can contain w_{ij} because w_{ij} has no "outgoing" dependency (*i.e.* co' contains no pair with w_{ij} as a first component), there is no cycle including some pair of transactions a_k, b_k and some pair y_{ij}, z_{ij} because there is no way to reach y_{ij} or z_{ij} from a_k or b_k, there is no cycle including only transactions a_k and b_k because a_{k_1} and b_{k_1} are not related to a_{k_2} and b_{k_2}, for $k_1 \neq k_2$, there is no cycle including transactions y_{i_1,j_1}, z_{i_1,j_1} and y_{i_2,j_2}, z_{i_2,j_2} for $i_1 \neq i_2$ since these are disconnected as well, and finally, there is no cycle including only transactions y_{ij} and z_{ij}, for a fixed i, because φ is satisfiable. It can be proved easily that the acyclic relation co' can be extended to a total commit order co which together with h_φ satisfies the $\mathsf{Serializability}$ axiom. Therefore, h_φ satisfies SER. □

3.3 Checking Consistency of Bounded-Width Histories

In this section, we show that checking prefix consistency, snapshot isolation, and serializability becomes polynomial time under the assumption that the *width* of the given history, i.e., the maximum number of mutually-unordered transactions w.r.t. the session order, is bounded by a fixed constant. If we consider the standard case where the session order is a union of transaction sequences (modulo the fictitious transaction writing the initial values), i.e., a set of sessions, then the width of the history is the number of sessions. We start by presenting an algorithm for checking serializability that is polynomial time when the width is bounded by a fixed constant. In general, the asymptotic complexity of this algorithm is exponential in the width of the history, but this worst-case behavior is not exercised in practice as shown in Section 3.5. Then, we prove that checking

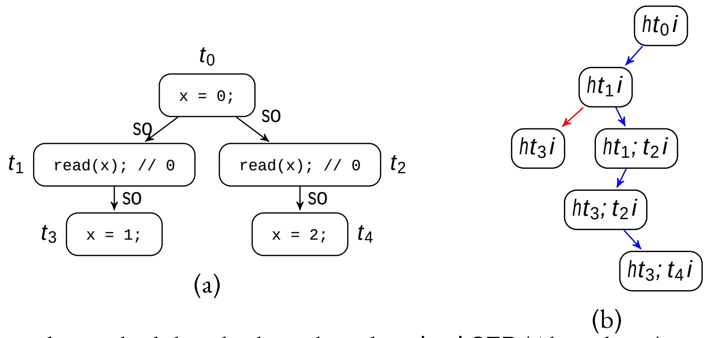


Figure 3.6: Applying the serializability checking algorithm checkSER (Algorithm 6) on the serializable history on the left. The right part pictures a search for valid extensions of serializable prefixes, represented by their boundaries. The red arrow means that the search is blocked (the prefix at the target is not a valid extension), while blue arrows mean that the search continues.

prefix consistency and snapshot isolation can be reduced in polynomial time to the problem of checking serializability.

3.3.1 Checking Serializability

We present an algorithm for checking serializability of a given history which constructs a valid commit order (satisfying Serialization), if any, by "linearizing" transactions one by one in an order consistent with the session order. At any time, the set of already linearized transactions is uniquely determined by an antichain of the session order (i.e., a set of mutually-unordered transactions w.r.t. so), and the next transaction to linearize is chosen among the immediate so successors of the transactions in this antichain. The crux of the algorithm is that the next transaction to linearize can be chosen such that it does not produce violations of Serialization in a way that does not depend on the order between the already linearized transactions. Therefore, the algorithm can be seen as a search in the space of so antichains. If the width of the history is bounded (by a fixed constant), then the number of possible so antichains is polynomial in the size of the history, which implies that the search can be done in polynomial time.

A *prefix* of a history $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$ is a set of transactions $T' \subseteq T$ such that all the so predecessors of transactions in T' are also in T', i.e., $\forall t \in T'.\ \mathsf{so}^{-1}(t) \in T$. A prefix T' is uniquely determined by the set of transactions in T' that are maximal w.r.t. so. This set of transactions forms an *antichain* of so, i.e., any two elements in this set are incomparable w.r.t. so. Given an antichain $\{t_1, \ldots, t_n\}$ of so, we say that $\{t_1, \ldots, t_n\}$ is the *boundary* of the prefix $T' = \{t : \exists i.\ \langle t, t_i \rangle \in \mathsf{so} \lor t = t_i\}$. For instance, given the history in Figure 3.6a, the set of transactions $\{t_0, t_1, t_2\}$ is a prefix with boundary $\{t_1, t_2\}$ (the latter is an antichain of the session order).

A prefix T' of a history h is called *serializable* iff there exists a *partial* commit order co on the transactions in h such that the following hold:

- co does not contradict the session order and the write-read relation in h, i.e., $\mathsf{wr} \cup \mathsf{so} \cup \mathsf{co}$ is acyclic,
- co is a total order on transactions in T',

- co orders transactions in T' before transactions in $T \setminus T'$, i.e., $\langle t_1, t_2 \rangle \in \mathsf{co}$ for every $t_1 \in T'$ and $t_2 \in T \setminus T'$,

- co does not order any two transactions $t_1, t_2 \notin T'$

- the history h along with the commit order co satisfies the axiom defining serializability, i.e., $\langle h, \mathsf{co} \rangle \models \mathsf{Serialization}$.

For the history in Figure 3.6a, the prefix $\{t_0, t_1, t_2\}$ is serializable since there exists a partial commit order co that orders t_0, t_1, t_2 in this order, and both t_1 and t_2 before t_3 and t_4. The axiom $\mathsf{Serialization}$ is satisfied trivially, since the prefix contains a single transaction writing x and all the transactions outside of the prefix do not read x.

A prefix $T' \uplus \{t\}$ of h is called a *valid extension*[8] of a serializable prefix T' of h, denoted by $T' \triangleright T' \uplus \{t\}$if:

- t does not read from a transaction outside of T', i.e., for every $t' \in T \setminus T'$, $\langle t', t \rangle \notin \mathsf{wr}$, and

- for every variable x written by t, there exists no transaction $t_2 \neq t$ outside of T' that reads a value of x written by a transaction t_1 in T', i.e., for every x written by t and every $t_1 \in T'$ and $t_2 \in T \setminus (T' \uplus \{t\})$, $\langle t_1, t_2 \rangle \notin \mathsf{wr}$.

For the history in Figure 3.6a, we have $\{t_0, t_1\} \triangleright \{t_0, t_1\} \uplus \{t_2\}$ because t_2 reads from t_0 and it does not write any variable. On the other hand $\{t_0, t_1\} \not\triangleright \{t_0, t_1\} \uplus \{t_3\}$ because t_3 writes x and the transaction t_2, outside of this prefix, reads from the transaction t_0 included in the prefix.

Let $\triangleright^*$ denote the reflexive and transitive closure of $\triangleright$.

The following lemma is essential in proving that iterative valid extensions of the initial empty prefix can be used to show that a given history is serializable.

Lemma 3.3.1. *For a serializable prefix T' of a history h, a prefix $T' \uplus \{t\}$ is serializable if it is a valid extension of T'.*

Proof. Let co' be the partial commit order for T' which satisfies the serializable prefix conditions. We extend co' to a partial order $\mathsf{co} = \mathsf{co}' \cup \{\langle t, t' \rangle | t' \notin T' \uplus \{t'\}\}$. We show that $\langle h, \mathsf{co} \rangle \models \mathsf{Serialization}$. The other conditions for $T' \uplus \{t\}$ being a serializable prefix are satisfied trivially by co.

Assume by contradiction that $\langle h, \mathsf{co} \rangle$ does not satisfy the axiom $\mathsf{Serialization}$. Then, there exists t_1, t_2, t_3, $x \in \mathsf{vars}(h)$ s.t. $\langle t_1, t_3 \rangle \in \mathsf{wr}_x$ and t_2 writes on x and $\langle t_1, t_2 \rangle, \langle t_2, t_3 \rangle \in \mathsf{co}$. Since $\langle h, \mathsf{co}' \rangle$ satisfies this axiom, at least one of these two co ordering constraints are of the form $\langle t, t' \rangle$ where $t' \notin T' \uplus \{t\}$:

- the case $t_1 = t$ and $t_2 \notin T' \uplus \{t\}$ is not possible because co' contains no pair of the form $\langle t', \_ \rangle \in \mathsf{co}'$ with $t' \notin T'$ (recall that $\langle t_2, t_3 \rangle$ should be also included in co).

- If $t_2 = t$ then, $\langle t_1, t_2 \rangle \in \mathsf{co}'$ and $\langle t_2, t_3 \rangle$ for some $t_3 \notin T' \uplus \{t\}$. But, by the definition of valid extension, for all variables x written by t, there exists no transaction $t_3 \notin T' \uplus \{t\}$ such that it reads x from $t_1 \in T'$. Therefore, this is also a contradiction.

Input: A history $h = (T, \mathsf{so}, \mathsf{wr})$, a serializable prefix T' of h
Output: *true* iff $T' \rhd^* h$

```
if T' = T then
    return true;
foreach t ∉ T' s.t. ∀t' ∈ T'. ⟨t', t⟩ ∈ wr ∪ so do
    if T' ⋫ T' ⊎ {t} then
        continue;
    if T' ⊎ {t} ∉ seen ∧ checkSER(h, T' ⊎ {t}) then
        return true;
    seen ← seen ∪ {(T' ⊎ {t})};
return false;
```

Algorithm 6: The algorithm checkSER for checking serializabilty. *seen* is a global variable storing a set of prefixes of h (which are not serializable). It is initialized as the empty set.

Algorithm 6 lists our algorithm for checking serializability. It is defined as a recursive procedure that searches for a sequence of valid extensions of a given prefix (initially, this prefix is empty) until covering the whole history. Figure 3.6b pictures this search on the history in Figure 3.6a. The right branch (containing blue edges) contains only valid extensions and it reaches a prefix that includes all the transactions in the history.

Theorem 3.3.1. *A history h is serializable iff $\mathsf{checkSER}(h, \emptyset)$ returns true.*

Proof. The "if" direction is a direct consequence of Lemma 3.3.1. For the reverse, assume that $h = \langle T, \mathsf{so}, \mathsf{wr}\rangle$ is serializable with a (total) commit order co. Let co_i be the set of transactions in the prefix of co of length i. Since co is consistent with so, we have that co_i is a prefix of h, for any i. We show by induction that co_{i+1} is a valid extension of co_i. The base case is trivial. For the induction step, let t be the last transaction in the prefix of co of length $i+1$. Then,

- t cannot read from a transaction outside of co_i because co is consistent with the write-read relation wr,
- also, for every variable x written by t, there exists no transaction $t_2 \neq t$ outside of co_i that reads a value of x written by a transaction $t_1 \in \mathsf{co}_i$. Otherwise, $\langle t_1, t_2\rangle \in \mathsf{wr}_x$, $\langle t, t_2\rangle \in \mathsf{co}$, and $\langle t_1, t\rangle \in \mathsf{co}$ which implies that $\langle h, \mathsf{co}\rangle$ does not satisfy Serializability.

This implies that $\mathsf{checkSER}(h, \emptyset)$ returns true. □

Algorithm 6 enumerates prefixes of the given history h, each prefix being uniquely determined by an antichain of h containing the so-maximal transactions in that prefix. By definition, the size of each antichain of a history h is smaller than the width of h. Therefore, the number of possible antichains (prefixes) of a history h is $O(\mathsf{size}(h)^{\mathsf{width}(h)})$ where $\mathsf{size}(h)$, resp., $\mathsf{width}(h)$, is the number of transactions, resp., the width, of h. Since the valid extension property can be checked

[8] We assume that $t \notin T'$ which is implied by the use of the disjoint union $\uplus$.

in quadratic time, the asymptotic time complexity of the algorithm defined by checkSER is upper bounded by $O(\mathsf{size}(h)^{\mathsf{width}(h)} \cdot \mathsf{size}(h)^3)$. The following corollary is a direct consequence of these observations.

Corollary 3.3.1. *For an arbitrary but fixed constant $k \in \mathbb{N}$, the problem of checking serializability for histories of width at most k is polynomial time.*

3.3.2 Reducing Prefix Consistency to Serializability

We describe a polynomial time reduction of checking prefix consistency of bounded-width histories to the analogous problem for serializability. Intuitively, as opposed to serializability, prefix consistency allows that two transactions read the same snapshot of the database and commit together even if they write on the same variable. Based on this observation, given a history h for which we want to check prefix consistency, we define a new history $h_{R|W}$ where each transaction t is split into a transaction performing all the reads in t and another transaction performing all the writes in t (the history $h_{R|W}$ retains all the session order and write-read dependencies of h). We show that if the set of read and write transactions obtained this way can be shown to be serializable, then the original history satisfies prefix consistency, and vice-versa. For instance, Figure 3.7 shows this transformation on the two histories in Figure 3.7a and Figure 3.7c, which represent typical anomalies known as "long fork" and "lost update", respectively. The former is not admitted by PC while the latter is admitted. It can be easily seen that the transformed history corresponding to the "long fork" anomaly is not serializable while the one corresponding to "lost update" is serializable. We show that this transformation leads to a history of the same width, which by Corollary 3.3.1, implies that checking prefix consistency of bounded-width histories is polynomial time.

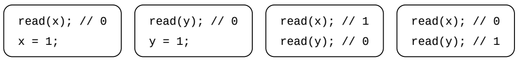
Thus, given a history $h = \langle T, \mathsf{wr}, \mathsf{so} \rangle$, we define the history $h_{R|W} = \langle T', \mathsf{wr}', \mathsf{so}' \rangle$ as follows:

- T' contains a transaction R_t, called a *read* transaction, and a transaction W_t, called a *write* transaction, for each transaction t in the original history, i.e., $T' = \{R_t | t \in T\} \cup \{W_t | t \in T\}$
- the write transaction W_t writes exactly the same set of variables as t, i.e., for each variable x, W_t writes to x iff t writes to x.
- the read transaction R_t reads exactly the same values and the same variables as t, i.e., for each variable x, $\mathsf{wr}_x{}' = \{\langle W_{t_1}, R_{t_2} \rangle | \langle t_1, t_2 \rangle \in \mathsf{wr}_x\}$
- the session order between the read and the write transactions corresponds to that of the original transactions and read transactions precede their write counterparts, i.e.,

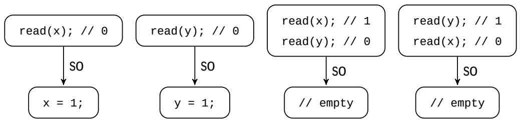
$$\mathsf{so}' = \{\langle R_t, W_t \rangle | t \in T\} \cup \{\langle R_{t_1}, R_{t_2} \rangle, \langle R_{t_1}, W_{t_2} \rangle, \langle W_{t_1}, R_{t_2} \rangle, \langle W_{t_1}, W_{t_2} \rangle | \langle t_1, t_2 \rangle \in \mathsf{so}\}$$

The following lemma is a straightforward consequence of the definitions.

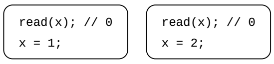
Lemma 3.3.2. *The histories h and $h_{R|W}$ have the same width.*



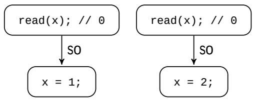
(a) Long fork



(b) Long fork (transformed)

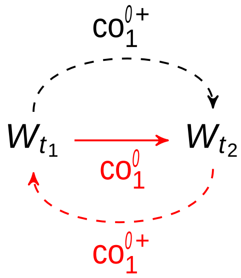


(c) Lost update

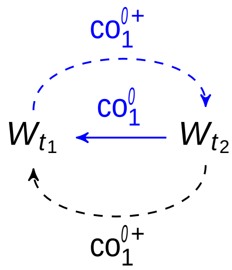


(d) Lost update (transformed)

Figure 3.7: Reducing PC to SER. Initially, the value of every variable is 0.



(a) $\langle W_{t_1}, W_{t_2}\rangle \in co_1'$



(b) $\langle W_{t_2}, W_{t_1}\rangle \in co_1'$

Figure 3.8: Cycles with non-consecutive write transactions.

Next, we show that $h_{R|W}$ is serializable if h is prefix consistent. Formally, we show that

$$\forall co.\ \exists co'.\ \langle h, co\rangle \models \text{Prefix} \Rightarrow \langle h_{R|W}, co'\rangle \models \text{Serializability}$$

Thus, let co be a commit (total) order on transactions of h which together with h satisfies the prefix consistency axiom. We define two *partial* commit orders co_1' and co_2', co_2' a strengthening of co_1', which we prove that they are acyclic and that any linearization co' of co_2' is a valid witness for $h_{R|W}$ satisfying serializability.

Thus, let co_1' be a *partial* commit order on transactions of $h_{R|W}$ defined as follows:

$$co_1' = \{\langle R_t, W_t\rangle \mid t \in T\} \cup \{\langle W_{t_1}, W_{t_2}\rangle \mid \langle t_1, t_2\rangle \in co\} \cup \{\langle W_{t_1}, R_{t_2}\rangle \mid \langle t_1, t_2\rangle \in wr \cup so\}$$

We show that if co_1' were to be cyclic, then it contains a minimal cycle with one read transaction, and at least one but at most two write transactions. Then, we show that such cycles cannot exist.

Lemma 3.3.3. *The relation co_1' is acyclic.*

Proof. We first show that if co_1' were to be cyclic, then it contains a minimal cycle with one read transaction, and at least one but at most two write transactions. Then, we show that such cycles cannot exist. Therefore, let us assume that co_1' is cyclic. Then,

- Since $\langle W_{t_1}, W_{t_2}\rangle \in co_1'$ implies $\langle t_1, t_2\rangle \in co$, for every t_1 and t_2, a cycle in co_1' cannot contain only write transactions. Otherwise, it will imply a cycle in the original commit order co. Therefore, a cycle in co_1' must contain at least one read transaction.

- Assume that a cycle in co'_1 contains two write transactions W_{t_1} and W_{t_2} which are not consecutive, like in Figure 3.8. Since either $\langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'_1$ or $\langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'_1$, there exists a smaller cycle in co'_1 where these two write transactions are consecutive. If $\langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'_1$, then co'_1 contains the smaller cycle on the lower part of the original cycle (Figure 3.8a), and if $\langle W_{t_2}, W_{t_1}\rangle \in \mathsf{co}'_1$, then co'_1 contains the cycle on the upper part of the original cycle (Figure 3.8b). Thus, all the write transactions in a minimal cycle of co'_1 must be consecutive.

- If a minimal cycle were to contain three write transactions, then all of them cannot be consecutive unless they all three form a cycle, which is not possible. So a minimal cycle contains at most two write transactions.

- Since co'_1 contains no direct relation between read transactions, it cannot contain a cycle with two consecutive read transactions, or only read transactions.

This shows that a minimal cycle of co'_1 would include a read transaction and a write transaction, and at most one more write transaction. We prove that such cycles are however impossible:

- if the cycle is of size 2, then it contains two transactions W_{t_1} and R_{t_2} such that $\langle W_{t_1}, R_{t_2}\rangle \in \mathsf{co}'_1$ and $\langle R_{t_2}, W_{t_1}\rangle \in \mathsf{co}'_1$. Since all the $\langle R\_, W\_\rangle$ dependencies in co'_1 are of the form $\langle R_t, W_t\rangle$, it follows that $t_1 = t_2$. Then, we have $\langle W_{t_1}, R_{t_1}\rangle \in \mathsf{co}'_1$ which implies $\langle t_1, t_1\rangle \in \mathsf{wr} \cup \mathsf{so}$, a contradiction.

- if the cycle is of size 3, then it contains three transactions W_{t_1}, W_{t_2}, and R_{t_3} such that $\langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'_1$, $\langle W_{t_2}, R_{t_3}\rangle \in \mathsf{co}'_1$, and $\langle R_{t_3}, W_{t_1}\rangle \in \mathsf{co}'_1$. Using a similar argument as in the previous case, $\langle R_{t_3}, W_{t_1}\rangle \in \mathsf{co}'_1$ implies $t_3 = t_1$. Therefore, $\langle t_1, t_2\rangle \in \mathsf{co}$ and $\langle t_2, t_1\rangle \in \mathsf{wr} \cup \mathsf{so}$, which contradicts the fact that $\mathsf{wr} \cup \mathsf{so} \subseteq \mathsf{co}$.

We define a strengthening of co'_1 where intuitively, we add all the dependencies from read transactions t_3 to write transactions t_2 that "overwrite" values read by t_3. Formally, $\mathsf{co}'_2 = \mathsf{co}'_1 \cup \mathsf{RW}(\mathsf{co}'_1)$ where

$$\mathsf{RW}(\mathsf{co}'_1) = \{\langle t_3, t_2\rangle \mid \exists x \in \mathsf{vars}(h).\ \exists t_1 \in T'.\ \langle t_1, t_3\rangle \in {\mathsf{wr}_x}',\ \langle t_1, t_2\rangle \in \mathsf{co}'_1,\ t_2 \text{ writes } x\}$$

It can be shown that any cycle in co'_2 would correspond to a Prefix violation in the original history. Therefore,

Lemma 3.3.4. *The relation co'_2 is acyclic.*

Proof. Assume that co'_2 is cyclic. Any minimal cycle in co'_2 still satisfies the properties of minimal cycles of co'_1 proved in Lemma 3.3.3 (because all write transactions are still totally ordered and co'_2 doesn't relate directly read transactions). So, a minimal cycle in co'_2 contains a read transaction and a write transaction, and at most one more write transaction.

Since co'_1 is acyclic, a cycle in co'_2, and in particular a minimal one, must necessarily contain a dependency from $\mathsf{RW}(\mathsf{co}'_1)$. Note that a minimal cycle cannot contain two such dependencies since this would imply that it contains two non-consecutive write transactions. The red edges in Figure 3.9a show a minimal cycle of co'_2 satisfying all the properties mentioned above. This cycle

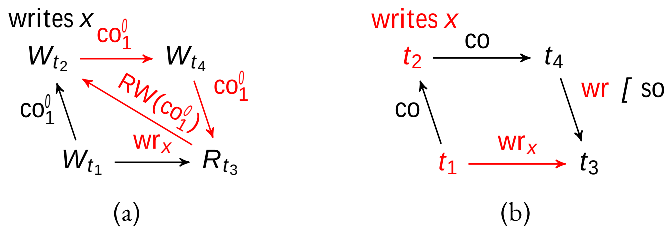


Figure 3.9: Cycles in co'_2 correspond to Prefix violations: (a) Minimal cycle in co'_2, (b) Prefix violation in $\langle h, \mathsf{co} \rangle$.

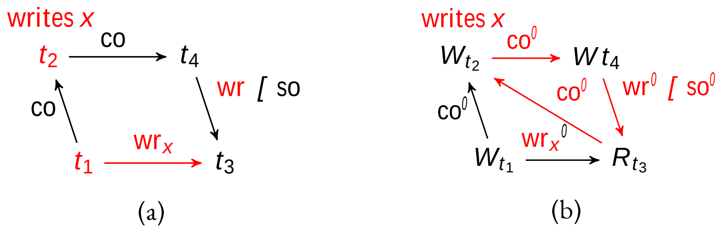


Figure 3.10: Prefix violations correspond to cycles in co': (a) Prefix violation in $\langle h, \mathsf{co} \rangle$, (b) Cycle in co'.

contains a dependency $\langle R_{t_3}, W_{t_2} \rangle \in \mathsf{RW}(\mathsf{co}'_1)$ which implies the existence of a write transaction W_{t_1} in $h_{R|W}$ s.t. $\langle W_{t_1}, R_{t_3} \rangle \in \mathsf{wr}_x{}'$ and $\langle W_{t_1}, W_{t_2} \rangle \in \mathsf{co}'_1$ and W_{t_1}, W_{t_2} write on x (these dependencies are represented by the black edges in Figure 3.9a). The relations between these transactions of $h_{R|W}$ imply that the corresponding transactions of h are related as shown in Figure 3.9b: $\langle W_{t_1}, W_{t_2} \rangle \in \mathsf{co}'_1$ and $\langle W_{t_2}, W_{t_4} \rangle \in \mathsf{co}'_1$ imply $\langle t_1, t_2 \rangle \in \mathsf{co}$ and $\langle t_2, t_4 \rangle \in \mathsf{co}$, respectively, $\langle W_{t_1}, W_{t_3} \rangle \in \mathsf{wr}_x{}'$ implies $\langle t_1, t_3 \rangle \in \mathsf{wr}_x$, and $\langle W_{t_4}, R_{t_3} \rangle \in \mathsf{co}'_1$ implies $\langle t_4, t_3 \rangle \in \mathsf{wr} \cup \mathsf{so}$. This implies that $\langle h, \mathsf{co} \rangle$ doesn't satisfy the Prefix axiom, a contradiction. □

Lemma 3.3.5. *If a history h satisfies prefix consistency, then $h_{R|W}$ is serializable.*

Proof. Let co' be any total order consistent with co'_2. Assume by contradiction that $\langle h_{R|W}, \mathsf{co}' \rangle$ doesn't satisfy $\mathsf{Serializability}$. Then, there exist $t'_1, t'_2, t'_3 \in T'$ such that $\langle t'_1, t'_2 \rangle, \langle t'_2, t'_3 \rangle \in \mathsf{co}'$ and t'_1, t'_2 write on some variable x and $\langle t'_1, t'_3 \rangle \in \mathsf{wr}_x{}'$. But then t'_1, t'_2 are write transactions and co'_1 must contain $\langle t'_1, t'_2 \rangle$. Therefore, $\mathsf{RW}(\mathsf{co}'_1)$ and co'_2 should contain $\langle t'_3, t'_2 \rangle$, a contradiction with co' being consistent with co'_2. □

Finally, it can be proved that any linearization co' of co'_2 satisfies $\mathsf{Serializability}$ (together with $h_{R|W}$). Moreover, it can also be shown that the serializability of $h_{R|W}$ implies that h satisfies PC. Therefore,

Theorem 3.3.2. *A history h satisfies prefix consistency iff $h_{R|W}$ is serializable.*

Proof. The "only-if" direction is proven by Lemma 3.3.5. For the reverse, we show that

$$\forall \mathsf{co}'.\ \exists \mathsf{co}.\ \langle h_{R|W}, \mathsf{co}' \rangle \models \mathsf{Serializability} \Rightarrow \langle h, \mathsf{co} \rangle \models \mathsf{Prefix}$$

Thus, let co' be a commit (total) order on transactions of $h_{R|W}$ which together with $h_{R|W}$ satisfies the serializability axiom. Let co be a commit order on transactions of h defined by $\mathsf{co} = \{\langle t_1, t_2\rangle \mid \langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'\}$ (co is clearly a total order). If co were not to be consistent with $\mathsf{wr} \cup \mathsf{so}$, then there would exist transactions t_1 and t_2 such that $\langle t_1, t_2\rangle \in \mathsf{wr} \cup \mathsf{so}$ and $\langle t_2, t_1\rangle \in \mathsf{co}$, which would imply that $\langle W_{t_1}, R_{t_2}\rangle, \langle R_{t_2}, W_{t_2}\rangle \in \mathsf{wr} \cup \mathsf{so}$ and $\langle W_{t_2}, W_{t_1}\rangle \in \mathsf{co}'$, which violates the acylicity of co'. We show that $\langle h, \mathsf{co}\rangle$ satisfies Prefix. Assume by contradiction that there exists a Prefix violation between t_1, t_2, t_3, t_4 (shown in Figure 3.10a), i.e., for some $x \in \mathsf{vars}(h)$, $\langle t_1, t_3\rangle \in \mathsf{wr}_x$ and t_2 writes x, $\langle t_1, t_2\rangle \in \mathsf{co}$, $\langle t_2, t_4\rangle \in \mathsf{co}^*$ and $\langle t_4, t_3\rangle \in \mathsf{wr} \cup \mathsf{so}$. Then, the corresponding transactions $W_{t_1}, W_{t_2}, W_{t_4}, R_{t_3}$ in $h_{R|W}$ would be related as follows: $\langle W_{t_1}, W_{t_2}\rangle \in \mathsf{co}'$ and $\langle W_{t_1}, R_{t_3}\rangle \in \mathsf{wr}_x'$ because $\langle t_1, t_3\rangle \in \mathsf{wr}_x$ and $\langle t_1, t_2\rangle \in \mathsf{co}$. Since co' satisfies $\mathsf{Serializability}$, then $\langle R_{t_3}, W_{t_2}\rangle \in \mathsf{co}'$. But $\langle t_2, t_4\rangle \in \mathsf{co}^*$ and $\langle t_4, t_3\rangle \in \mathsf{wr} \cup \mathsf{so}$ imply that $\langle W_{t_2}, W_{t_4}\rangle \in \mathsf{co}'^*$ and $\langle W_{t_4}, R_{t_3}\rangle \in \mathsf{wr}' \cup \mathsf{so}'$, which show that co' is cyclic (the red cycle in Figure 3.10b), a contradiction.

Since the history $h_{R|W}$ can be constructed in linear time, Lemma 3.3.2, Theorem 3.3.2, and Corollary 3.3.1 imply the following result.

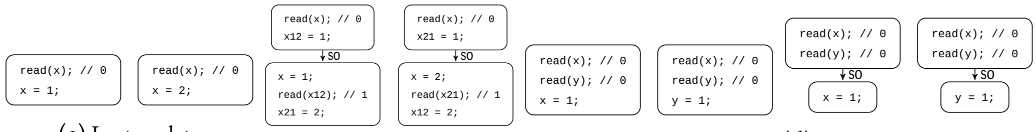
Corollary 3.3.2. *For an arbitrary but fixed constant $k \in \mathbb{N}$, the problem of checking prefix consistency for histories of width at most k is polynomial time.*

3.3.3 Reducing Snapshot Isolation to Serializability

We extend the reduction of prefix consistency to serializability to the case of snapshot isolation. Compared to prefix consistency, snapshot isolation disallows transactions that read the same snapshot of the database to commit together if they write on a common variable (stated by the $\mathsf{Conflict}$ axiom). More precisely, for any pair of transactions t_1 and t_2 writing to a common variable, t_1 must observe the effects of t_2 or vice-versa. We refine the definition of $h_{R|W}$ such that any "serialization" (i.e.., commit order satisfying $\mathsf{Serializability}$) disallows that the read transactions corresponding to two such transactions are ordered both before their write counterparts. We do this by introducing auxiliary variables that are read or written by these transactions. For instance, Figure 3.11 shows this transformation on the two histories in Figure 3.11a and Figure 3.11c, which represent the anomalies known as "lost update" and "write skew", respectively. The former is not admitted by SI while the latter is admitted. Concerning "lost update", the read counterpart of the transaction on the left writes to a variable `x12` that is read by its write counterpart, but also written by the write counterpart of the other transaction. This forbids that the latter is serialized in between the read and write counterparts of the transaction on the left. A similar scenario is imposed on the transaction on the right, which makes that the transformed history is not serializable. Concerning the "write skew" anomaly, the transformed history is exactly as for the PC reduction since the two transactions don't write on a common variable. It is clearly serializable.

For a history $h = \langle T, \mathsf{wr}, \mathsf{so}\rangle$, the history $h^c_{R|W} = \langle T', \mathsf{wr}', \mathsf{so}'\rangle$ is defined as $h_{R|W}$ with the following additional construction: for every two transactions t_1 and $t_2 \in T$ that write on a common variable,

- R_{t_1} and W_{t_2} (resp., R_{t_2} and W_{t_1}) write on a variable $x_{1,2}$ (resp., $x_{2,1}$),
- the write transaction of t_i reads $x_{i,j}$ from the read transaction of t_i, for all $i \neq j \in \{1, 2\}$, i.e., $\mathsf{wr}_{x_{1,2}} = \{\langle R_{t_1}, W_{t_1}\rangle\}$ and $\mathsf{wr}_{x_{2,1}} = \{\langle R_{t_2}, W_{t_2}\rangle\}$.



(a) Lost update

(b) Lost update (transformed)

(c) Write skew

(d) Write skew (transformed)

Figure 3.11: Reducing SI to SER.

Note that h_{RjW} and h^c_{RjW} have the same width (the session order is defined exactly in the same way), which implies, by Lemma 3.3.2, that h and h^c_{RjW} have the same width.

The following result can be proved using similar reasoning as in the case of prefix consistency.

Theorem 3.3.3. *A history h satisfies snapshot isolation iff h^c_{RjW} is serializable.*

Note that h^c_{RjW} and h have the same width, and that h^c_{RjW} can be constructed in linear time. Therefore, Theorem 3.3.3, and Corollary 3.3.1 imply the following result.

Corollary 3.3.3. *For an arbitrary but fixed constant $k \in \mathbb{N}$, the problem of checking snapshot isolation for histories of width at most k is polynomial time.*

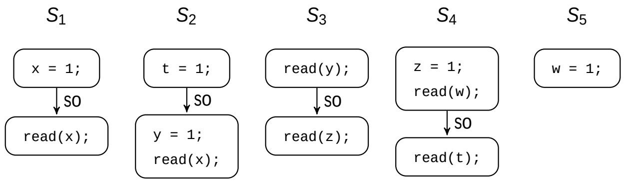
3.4 Communication graphs

In this section, we present an extension of the polynomial time results for PC, SI, and SER, which allows to handle histories where the sharing of variables between different sessions is *sparse*. For the results in this section, we take the simplifying assumption that the session order is a union of transaction sequences (modulo the fictitious transaction writing the initial values), i.e., each transaction sequence corresponding to the standard notion of *session*[9]. We represent the sharing of variables between different sessions using an undirected graph called a *communication graph*. For instance, the communication graph of the history in Figure 3.12a is given in Figure 3.12b. For readability, the edges are marked with the variables accessed by the two sessions.

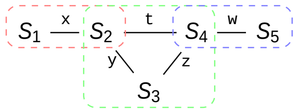
We show that the problem of checking PC, SI, or SER is polynomial time when the size of every *biconnected* component of the communication graph is bounded by a fixed constant. This is stronger than the results in Section 3.3 because the number of biconnected components can be arbitrarily large which means that the total number of sessions is unbounded. In general, we prove that the time complexity of these consistency criteria is exponential only in the maximum size of such a biconnected component, and not the whole number of sessions.

An undirected graph is biconnected if it is connected and if any one vertex were to be removed, the graph will remain connected, and a biconnected component of a graph G is a maximal biconnected subgraph of G. Figure 3.12b shows the decomposition in biconnected components of a communication graph. This graph contains 5 sessions while every biconnected component is of size at most 3. Intuitively, if a history h is a violation to some consistency criterion $C \in \{\mathrm{PC}, \mathrm{SI}, \mathrm{SER}\}$, then there exists a projection of h on sessions from the *same* biconnected

[9] The results can be extended to arbitrary session orders by considering maximal transaction sequences in session order instead of sessions.



(a) A history with 5 sessions.



(b) The communication graph and its decomposition in biconnected components.

Figure 3.12: A history and its communication graph.

component which is also a violation to C (the reverse is trivially true). Therefore, checking any of these criteria can be done in isolation for each biconnected component (more precisely, on sub-histories that contain only sessions in the same biconnected component). Actually, this decomposition argument works even for RC, RA, and CC. For instance, in the case of the history in Figure 3.12a, any consistency criterion can be checked looking in isolation at three sub-histories: a sub-history with S_1 and S_2, a sub-history with S_2, S_3, and S_4, and a sub-history with S_4 and S_5.

Formally, a *communication graph* of a history h is an undirected graph $\mathsf{Comm}(h) = (V, E)$ where the set of vertices V is the set of sessions[10] in h, and $(v, v') \in E$ iff the sessions v and v' contain two transactions t_1 and t_2, respectively, such that t_1 and t_2 read or write a common variable x.

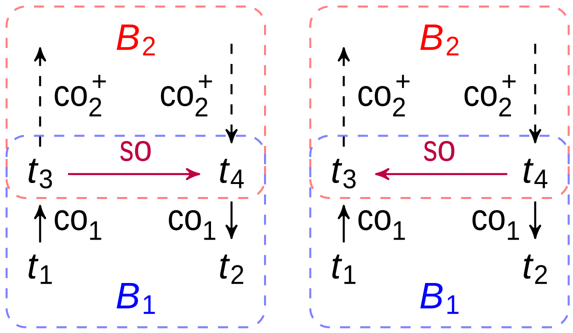
We begin with a technical lemma showing that *minimal* paths of certain form in the graph representing a history h and a relation co (on the transactions of h) lie within a single biconnected component of the underlying communication graph. This is used to show that any consistency violation can be exposed by looking at a single biconnected component at a time. The graph representing a history h and a relation co on the transactions of h is denoted by $\mathsf{G}(h, \mathsf{co})$[11].

Given a graph $\mathsf{G}(h, \mathsf{co})$ and a relation r on its vertices, a term over the relations so, wr, and co, e.g., $(\mathsf{wr} \cup \mathsf{so})^+$, a path of the form r (or an r-path) is a sequence of edges representing so, wr, or co dependencies as specified by the term r, e.g., a sequence of wr or so dependencies.

Lemma 3.4.1. *Let $B_1, \ldots, B_n$ be the biconnected components of $\mathsf{Comm}(h)$ for a history $h = \langle T, \mathsf{wr}, \mathsf{so} \rangle$. For each B_i, let co_i be a total order on the transactions of B_i*[12] *extending the session order so on the transactions of B_i. Also, let $\mathsf{co} = \bigcup_i \mathsf{co}_i$. Then, for every term $r \in \{\mathsf{co}^+, (\mathsf{wr} \cup \mathsf{so})^+\}$, any minimal r-path in the graph $\mathsf{G}(h, \mathsf{co})$ between two transactions from the same biconnected component includes only transactions of that biconnected component.*

Proof. We consider the case $r = \mathsf{co}^+$. Consider a *minimal* co^+-path $\pi = t_0, \ldots, t_n$ between two transactions t_0 and t_n from the same biconnected component B of $\mathsf{Comm}(h)$ (i.e., from sessions in B). Assume by contradiction, that π traverses multiple biconnected components. We define a path $\pi_s = v_0, \ldots, v_m$ between sessions, i.e., vertices of $\mathsf{Comm}(h)$, which contains an edge (v_j, v_{j+1}) iff π contains an edge (t_i, t_{i+1}) with t_i a transaction of session v_j and t_{i+1} a transaction of session $v_{j+1} \neq v_j$. Since any graph decomposes to a forest of biconnected components, this path must necessarily leave and enter some biconnected component B_1 to and from

[10] The transaction writing the initial values is considered as a distinguished session.
[11] The nodes of $\mathsf{G}(h, \mathsf{co})$ correspond to transactions in h and the edges connect pairs of transactions in so, wr, or co.
[12] That is, transactions that are included in the sessions in B_i.



(a) $\langle t_3, t_4\rangle \in so$ (b) $\langle t_4, t_3\rangle \in so$

Figure 3.13: Minimal paths between transactions in the same biconnected component.

the same biconnected component B_2, i.e., π_s must contain two vertices v_{j_1} and v_{j_2} in B_1 such that the successor v_{j_1+1} of v_{j_1} and the predecessor v_{j_2-1} of v_{j_2} are from B_2. Let t_1, t_2, t_3, t_4 be the transactions in the path π corresponding to v_{j_1}, v_{j_2}, v_{j_1+1}, and v_{j_2-1}, respectively. Now, since any two biconnected components share at most one vertex, it follows that t_3 and t_4 are from the same session and

- if $\langle t_3, t_4\rangle \in so$, then there exists a shorter path between t_0 and t_1 that uses the so relation between $\langle t_3, t_4\rangle$ (we recall that $so \subseteq \bigcup_i co_i$) instead of the transactions in B_2, pictured in Figure 3.13a, which is a contradiction to the minimality of π,
- if $\langle t_4, t_3\rangle \in so$, then, we have a cycle in $\bigcup_i co_i \cup so$, pictured in Figure 3.13b, which is also a contradiction.

The case $r = (wr \cup so)^+$ can be proved in a similar manner since the reasoning outlined in Figure 3.13 reduces to short-circuiting a path using a single so edge (and so is included in $(wr \cup so)^+$).

Now we prove our final claim. For a history $h = (T, so, wr)$ and biconnected component B of $\mathsf{Comm}(h)$, the projection of h over transactions in sessions of B is denoted by $h \downarrow B$, i.e., $h \downarrow B = (T', so', wr')$ where T' is the set of transactions in sessions of B, so' and wr' are the projections of so and wr, respectively, on T'.

Theorem 3.4.1. *For any criterion $C \in \{RA, RC, CC, PC, SI, SER\}$, a history h satisfies C iff for every biconnected component B of $\mathsf{Comm}(h)$, $h \downarrow B$ satisfies C.*

Proof. The "only-if" direction is obvious. For the "if" direction, we first consider the cases $C \in \{RA, RC, CC, SER\}$. The proof concerning PC and SI is based on the reduction to SER outlined in Section 3.3.2 and Section 3.3.3, respectively, and it is given afterwards. Let $B_1, \ldots, B_n$ be the biconnected components of $\mathsf{Comm}(h)$.

Let $C \in \{RA, RC, CC, SER\}$, and let co_i be the commit order that witnesses that $h \downarrow B_i$ satisfies C, for each $1 \leq i \leq n$. The union $\bigcup_i co_i$ is acyclic since otherwise, any minimal cycle would be a minimal path between transactions of the same biconnected component B_j, and, by Lemma 3.4.1, it will include only transactions of B_j which is a contradiction to co_j being a total order. We show that any linearization co of $\bigcup_i \mathsf{co}_i$ along with h satisfies the axioms of C. The axioms defining RA, RC, CC, and SER involve transactions that write or read a common variable, which implies that they belong to the same biconnected component (we refer to the transactions t_1, t_2, and t_3 in Figure 3.2). Furthermore, by Lemma 3.4.1, minimal paths witnessing the dependencies in those axioms, e.g., $(\mathsf{wr} \cup \mathsf{so})^+$ for CC, are also formed of transactions included in the same biconnected component. Therefore, co satisfies any of those axioms provided that each co_i does.

We now consider the case where C = PC. Assume that each B_i satisfies PC. Based on the reduction in Section 3.3.2, h satisfies PC iff $h_{R|W}$ satisfies SER. Moreover, since $h_{R|W}$ is obtained from h by splitting each transaction t into a read transaction R_t and a write transaction W_t while keeping all session order dependencies, each session in h corresponds to a session in $h_{R|W}$ that reads or writes exactly the same set of variables. Therefore, $\mathsf{Comm}(h)$ is isomorphic to $\mathsf{Comm}(h_{R|W})$. Since B_i satisfies PC, we get that the corresponding biconnected component B_i' of $\mathsf{Comm}(h_{R|W})$ satisfies SER, for every i. Therefore, $h_{R|W}$ satisfies SER, which implies that h satisfies PC. The case of SI is proved in a similar way using the reduction to the serializability of $h^c_{R|W}$ presented in Section 3.3.3 (note that two transactions of $h^c_{R|W}$ may read or write an additional common variable only if they were writing a common variable in the original history and therefore, $\mathsf{Comm}(h)$ is still isomorphic to $\mathsf{Comm}(h^c_{R|W})$). □

Since the decomposition of a graph into biconnected components can be done in linear time, Theorem 3.4.1 implies that any of the criteria PC, SI, or SER can be checked in time $O(\mathsf{size}(h)^{\mathsf{bi\text{-}size}(h)} \cdot \mathsf{size}(h)^3 \cdot \mathsf{bi\text{-}nb}(h))$ where $\mathsf{bi\text{-}size}(h)$ and $\mathsf{bi\text{-}nb}(h)$ are the maximum size of a biconnected component in $\mathsf{Comm}(h)$ and the number of biconnected components of $\mathsf{Comm}(h)$, respectively. The following corollary is a direct consequence of this observation.

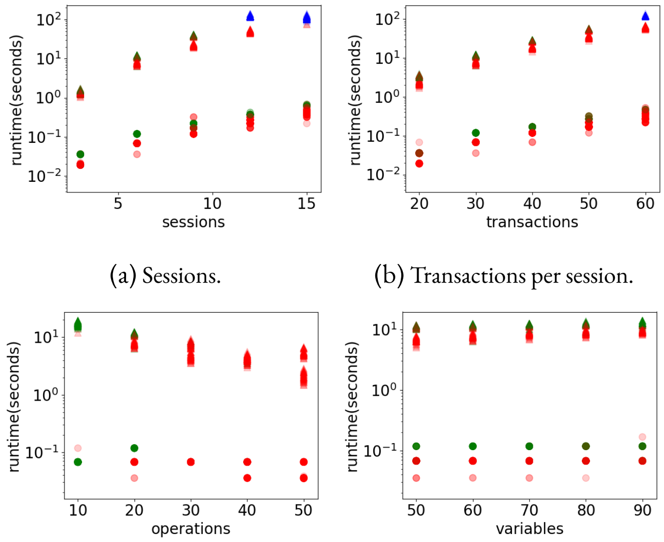
Corollary 3.4.1. *For an arbitrary but fixed constant $k \in \mathbb{N}$ and any criterion $C \in \{PC, SI, SER\}$, the problem of checking if a history h satisfies C is polynomial time, provided that the size of every biconnected component of $\mathsf{Comm}(h)$ is bounded by k.*

3.5 Experimental Evaluation

To demonstrate the practical value of the theory developed in the previous sections, we argue that our algorithms:

- are efficient and scalable,
- enable an effective testing framework allowing to expose consistency violations in production databases.

We focus on three of the criteria introduced in Section 3.1: *serializability* which is NP-complete in general and polynomial time when the number of sessions is considered to be a constant, *snapshot isolation* which can be reduced in linear time to serializability, and *causal consistency* which is polynomial time in general. As benchmark, we consider histories extracted from three distributed databases: CockroachDB [34], Galera [49], and AntidoteDB [6]. Following the approach in Jepsen [63], histories are generated with random clients. For the experiments described hereafter, the randomization process is parametrized by: (1) the number of sessions (**#sess**), (2) the



(a) Sessions. (b) Transactions per session.

(c) Operations per transaction. (d) Variables.

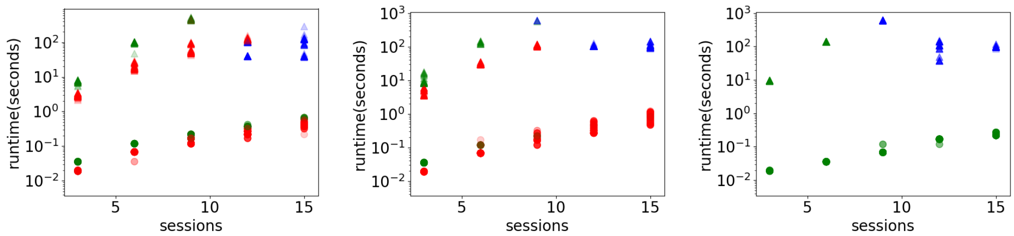
Figure 3.14: Scalability of our algorithm for checking Serializability (Algorithm 6) with comparison to a SAT encoding. The x-axis represents the varying parameter while the y-axis represents the wall-clock time in logarithmic scale. The circular, resp., triangular, dots represent wall-clock times of our algorithm, resp., the SAT encoding. The red, green, and blue dots represent invalid, valid and resource-exhausted instances, respectively.

number of transactions per session (**#trs**), (3) the number of operations per transaction (**#ops**), and (4) an upper bound on the number of used variables (**#vars**)[13]. For any valuation of these parameters, half of the histories generated with CockroachDB and Galera are restricted such that the sets of variables written by any two sessions are disjoint (the sets of read variables are not constrained). This restriction is used to increase the frequency of valid histories.

In a first experiment, we investigated the efficiency of our serializability-checking algorithm (Algorithm 6) and we compared its performance with a direct SAT encoding[14] of the serializability definition in Section 3.1 (we used MiniSAT [40] to solve the SAT queries). We used histories extracted from CockroachDB which claims to implement serializability, acknowledging however the possibility of anomalies [35]. The sessions of a history are uniformly distributed among 3 nodes of a single cluster. To evaluate scalability, we fix a reference set of parameter values: **#sess**=6, **#trs**=30, **#ops**=20, and **#vars** = 60 **#sess**, and vary only one parameter at a time. For instance, the number of sessions varies from 3 to 15 in increments of 3. We consider 100 histories for each combination of parameter values. The experimental data is reported in Figure 3.14. Our algorithm scales well even when increasing the number of sessions, which is not guaranteed by its

[13] We ensure that every value is written at most once.

[14] For each ordered pair of transactions t_1, t_2 we add two propositional variables representing $\langle t_1, t_2\rangle \in (\mathsf{wr} \cup \mathsf{so})^+$ and $\langle t_1, t_2\rangle \in \mathsf{co}$, respectively. Then we generate clauses corresponding to: (1) singleton clauses defining the relation $\mathsf{wr} \cup \mathsf{so}$ (extracted from the input history), (2) $\langle t_1, t_2\rangle \in \mathsf{wr} \cup \mathsf{so}$ implies $\langle t_1, t_2\rangle \in \mathsf{co}$, (3) co being a total order, and (4) the axioms corresponding to the considered consistency model. This is an optimization that does not encode wr and so separately, which is sound because of the shape of our axioms (and because these relations are fixed apriori).



(a) Checking SI (CockroachDB) (b) Checking SI (Galera) (c) Checking CC (AntidoteDB)

Figure 3.15: Scalability of our algorithms for checking Snapshot isolation (Section 3.3.3) and Causal consistency (Algorithm 5) with comparison to a SAT encoding. The x-axis represents the varying parameter while the y-axis represents the wall-clock time in logarithmic scale. The circular, resp., triangular, dots represent wall-clock times of our algorithm, resp., the SAT encoding. The red, green, and blue dots represent invalid, valid and resource-exhausted instances, respectively.

worst-case complexity (in general, this is exponential in the number of sessions). Also, our algorithm is at least two orders of magnitude more efficient than the SAT encoding. While the performance of SAT solvers is known to be heavily affected by the specific encoding of the problem, we strove to make the SAT formula as succinct as possible and optimize its construction. We have fixed a 10 minutes timeout, a limit of 10GB of memory, and a limit of 10GB on the files containing the formulas to be passed to the SAT solver. The blue dots represent resource-exhausted instances. The SAT encoding reaches the file limit for 148 out of 200 histories with at least 12 sessions (Figure 3.14a) and for 50 out of 100 histories with 60 transactions per session (Figure 3.14b), the other parameters being fixed as explained above.

We have found a large number of violations, whose frequency increases with the number of sessions, transactions per session, or operations per transaction, and decreases when allowing more variables. This is expected since increasing any of the former parameters increases the chance of interference between different transactions while increasing the latter has the opposite effect. The second and third column of Table 3.2 give a more precise account of the kind of violations we found by identifying for each criterion X, the number of histories that violate X but no other criterion weaker than X, e.g., there is only one violation to SI that satisfies PC.

The second experiment measures the scalability of the SI checking algorithm obtained by applying the reduction to SER described in Section 3.3.3 followed by the SER checking algorithm in Algorithm 6, and its performance compared to a SAT encoding of SI. Actually, the reduction to SER is performed on-the-fly, while traversing the history and checking for serializability (of the transformed history). The SAT encoding follows the same principles as in the case of serializability. We focus on its behavior when increasing the number of sessions (varying the other parameters leads to similar results). As benchmark, we used the same CockroachDB histories as in Figure 3.14a and a number of histories extracted from Galera[15] whose documentation contains contradicting claims about whether it implements snapshot isolation [50, 51]. We use 100 histories per combination of parameter values as in the previous experiment. The results are reported

[15] In order to increase the frequency of valid histories, all sessions are executed on a single node.

Table 3.2: Violation statistics. The "disjoint writes" columns refer to histories where the set of variables written by any two sessions are disjoint.

	Serializability checking		Snapshot Isolation checking	
Weakest criterion violated	CockroachDB (disjoint writes)	CockroachDB (no constraints)	Galera (disjoint writes)	Galera (no constraints)
Read Committed			19	50
Read Atomic	180	547	91	139
Causal Consistency	339	382	88	43
Prefix Consistency	2	7		
Snapshot Isolation		1		1
Serializability	25			
Total number of violations	546/1000	937/1000	198/250	233/250

in Figure 3.15a and Figure 3.15b. We observe the same behavior as in the case of SER. In particular, the SAT encoding reaches the file limit for 150 out of 200 histories with at least 12 sessions in the case of the CockroachDB histories, and for 162 out of 300 histories with at least 9 sessions in the case of the Galera histories. The last two columns in Table 3.2 classify the set of violations depending on the weakest criterion that they violate.

We also evaluated the performance of the CC checking algorithm in Section 3.2 when increasing the number of sessions, on histories extracted from AntidoteDB, which claims to implement causal consistency [7]. The results are reported in Figure 3.15c. In this case, the SAT encoding reaches the file limit for 150 out of 300 histories with at least 9 sessions. All the histories considered in this experiment are valid. However, when experimenting with other parameter values, we have found several violations. The smallest parameter values for which we found violations were 3 sessions, 14 transactions per session, 14 operations per transaction, and 5 variables. The violations we found are also violations of Read Atomic. For instance, one of the violations contains two transactions t_1 and t_2, each of them writing to two variables x_1 and x_2, and another transaction t_3 that reads x_1 from t_1 and x_2 from t_2 (t_1 and t_2 are from different sessions while t_3 is an so successor of t_1 in the same session). These violations are novel and they were confirmed by the developers of AntidoteDB.

The refinement of the algorithms above based on communication graphs, described in Section 3.4, did not have a significant impact on their performance. The histories we generated contained few biconnected components (many histories contained just a single biconnected component) which we believe is due to our proof of concept deployment of these databases on a single machine that did not allow to experiment with very large number of sessions and variables.

3.6 Related Work

[31] give the first formalization of the criteria we consider in this work, using the specification methodology of [29]. This formalization uses two auxiliary relations, a *visibility* relation which represents the fact that a transaction "observes" the effects of another transaction and a *commit order*, also called arbitration order, like in our case. Executions are abstracted using a notion of history that includes only a session order and the adherence to some consistency criterion is defined

as the existence of a visibility relation and a commit order satisfying certain axioms. Motivated by practical goals, our histories include a write-read relation, which enables more uniform and in our opinion, more intuitive, axioms to characterize consistency criteria. Our formalizations are however equivalent with those of [31] (a formal proof of this equivalence is presented in the extended version of this paper [18]). Moreover, [31] do not investigate algorithmic issues as in our work.

[80] showed that checking serializability of an execution is NP-complete. Moreover, it identifies a stronger criterion called *conflict serializability* which is polynomial-time checkable. Conflict serializability assumes that histories are given as sequences of operations and requires that the commit order be consistent with a *conflict-order* between transactions defined based on this sequence (roughly, a transaction t_1 is before a transaction t_2 in the conflict order if it accesses some variable x before t_2 does). This result is not applicable to distributed databases where deriving such a sequence between operations submitted to different nodes in a network is impossible.

[21] showed that checking several variations of causal consistency on executions of a *non-transactional* distributed database is polynomial time (they also assume that every value is written at most once). Assuming singleton transactions, our notion of CC corresponds to the causal convergence criterion in [21]. Therefore, our result concerning CC can be seen as an extension of this result concerning causal convergence to transactions.

There are some works that investigated the problem of checking consistency criteria like sequential consistency and linearizability in the case of shared-memory systems. [54] showed that checking linearizability of the single-value register type is NP-complete in general, but polynomial time for executions where every value is written at most once. Using a reduction from serializabilty, they showed that checking sequential consistency is NP-complete even when every value is written at most once. [42] extended the result concerning linearizability to a series of abstract data types called collections, that includes stacks, queues, key-value maps, etc. Sequential consistency reduces to serializability for histories with singleton transactions (i.e., formed of a single read or write operation). Therefore, our polynomial-time result for checking serializability of bounded-width histories (Corollary 3.3.1) implies that checking sequential consistency of histories with a bounded number of threads is polynomial time. The latter result has been established independently by [2].

The notion of *communication graph* is inspired by the work of [33], which investigates partial-order reduction (POR) techniques for multi-threaded programs. In general, the goal of partial-order reduction [48] is to avoid exploring executions which are equivalent w.r.t. some suitable notion of equivalence, e.g., Mazurkiewicz trace equivalence [73]. They use the acyclicity of communication graphs to define a class of programs for which their POR technique is optimal. The algorithmic issues they explore are different than ours and they don't investigate biconnected components of this graph as in our results.

3.7 Conclusion

In this chapter, we proposed novel logical characterizations of various consistency models of transactional systems such as *Read Committed* (RC) [12], *Read Atomic* (RA) [31], *Causal Consistency* (CC) [68], *Prefix Consistency* (PC) [30], *Snapshot Isolation* (SI) [12] and Serializability (SER). This enables an investigation of algorithmic techniques for checking conformance of a given execution. We establish the asymptotic complexity of this problem for each consistency model when the read-from relation is known apriori. We introduce polynomial-time algorithms for RC, RA, CC, and prove that checking conformance for PC, SI, and SER is NP-complete. In the latter case, we introduce conformance checking algorithms that are polynomial time when the number of bi-connected components of the communication graph is bounded by a fixed constant. Finally, we demonstrate a runtime performance analysis of an implementation of our algorithms based on histories from production databases. As benchmark, we consider histories extracted from three distributed databases: CockroachDB [34], Galera [49], and AntidoteDB [6]. Using our implementation, we were able to find bugs in all these distributed databases, which confirms their incorrect promises on strong guarantees (CockroachDB) or previously mentioned bugs (Galera) or novel bugs (AntidoteDB).

4 Testing Applications That Use Transactional Data Stores

We present MonkeyDB, a mock in-memory storage system meant for testing correctness of storage-backed applications. MonkeyDB supports common APIs for accessing data (key-value updates, as well as SQL queries), making it an easy substitute for an actual storage system. MonkeyDB can be configured with one of several transaction isolation (consistency) levels.

MonkeyDB implements a *centralized* operational semantics for key-value stores, which is based on the axiomatic definitions presented in Section 3.1. Transactions are executed *serially*, one after another, the concurrency being simulated during the handling of read events. This semantics maintains a history that contains all the past events (from all transactions/sessions), and write events are simply added to the history. The value returned by a read event is established based on a non-deterministic choice of a write-read dependency (concerning this read event) that satisfies the axioms of the considered consistency models. Depending on the weakness of the isolation level, this makes it possible to return values written in arbitrarily "old" transactions, and simulate any concurrent behavior.

We formally prove that this semantics does indeed simulate any concurrent behavior, by showing that it is equivalent to a semantics where transactions are allowed to interleave. In comparison with concrete implementations, this semantics makes it possible to handle a wide range of consistency models in a uniform way. It only has two sources of non-determinism: the order in which entire transactions are submitted, and the choice of write-read dependencies in read events. This enable better coverage of possible behaviors, the penalty in performance not being an issue in safety testing workloads which are usually small (see our evaluation).

We also extend our semantics to cover SQL queries as well, by compiling SQL queries down to transactions with multiple key-value reads/writes. A table in a relational database is represented using a set of primary key values (identifying uniquely the set of rows) and a set of keys, one for each cell in the table. The set of primary key values is represented using a set of Boolean key-value pairs that simulate its characteristic function (adding or removing an element corresponds to updating one of these keys to `true` or `false`). Then, SQL queries are compiled to read or write accesses to the keys representing a table. For instance, a `SELECT` query that retrieves the set of rows in a table that satisfy a `WHERE` condition is compiled to (1) reading Boolean keys to identify the primary key values of the rows contained in the table, (2) reading keys that represent columns used in the `WHERE` condition, and (3) reading all the keys that represent cells in a row satisfying the `WHERE` condition. This rewriting contains the minimal set of accesses to the cells of a table that are needed to ensure the conventional specification of SQL. It makes it possible to "export" formalizations of key-value store consistency models to SQL transactions.

The remainder of this chapter is organized as follows:

$$k \in \mathsf{Keys} \quad x \in \mathsf{Vars} \quad tab \in \mathsf{T} \quad c, c_1, c_2 \in \mathsf{C}$$

$$\begin{aligned}
\mathsf{Prog} &\stackrel{\text{def}}{=} \mathsf{Sess} \mid \mathsf{Sess} \,||\, \mathsf{Prog} \\
\mathsf{Sess} &\stackrel{\text{def}}{=} \mathsf{Trans} \mid \mathsf{Trans}; \mathsf{Sess} \\
\mathsf{Trans} &\stackrel{\text{def}}{=} \mathtt{begin}; \mathsf{Body}; \mathtt{commit} \\
\mathsf{Body} &\stackrel{\text{def}}{=} \mathsf{Instr} \mid \mathsf{Instr}; \mathsf{Body} \\
\mathsf{Instr} &\stackrel{\text{def}}{=} \mathsf{InstrKV} \mid \mathsf{InstrSQL} \mid x := e \mid \mathtt{if}(\ (x))\{\mathsf{Instr}\} \\
\mathsf{InstrKV} &\stackrel{\text{def}}{=} x := \mathtt{read}(k) \mid \mathtt{write}(k, x) \\
\mathsf{InstrSQL} &\stackrel{\text{def}}{=} \mathtt{SELECT}\ c_1\ \mathtt{AS}\ x\ \mathtt{FROM}\ tab\ \mathtt{WHERE}\ \ (c_2) \mid \\
&\qquad \mathtt{INSERT\ INTO}\ tab\ \mathtt{VALUES}\ x \mid \\
&\qquad \mathtt{DELETE\ FROM}\ tab\ \mathtt{WHERE}\ \ (c) \mid \\
&\qquad \mathtt{UPDATE}\ tab\ \mathtt{SET}\ c_1 = x\ \mathtt{WHERE}\ \ (c_2)
\end{aligned}$$

Figure 4.1: Program syntax. The set of all keys is denoted by Keys, Vars denotes the set of local variables, T the set of table names, and C the set of column names. We use to denote Boolean expressions, and e to denote expressions interpreted as values. We use ~ to denote vectors of elements.

- Section 4.1 presents a programming language to represent storage-backed applications,
- Section 4.2 defines an operational semantics for key-value stores under various consistency models, which simulates all concurrent behaviors with executions where transactions execute serially and which is based on the axiomatic definitions in Section 3.1,
- Section 4.3 broadens the scope of the key-value store semantics to SQL transactions using a compiler that rewrites SQL queries to key-value accesses,
- Section 4.4 describes the implementation of MonkeyDB,
- Section 4.5 and Section 4.6 present an evaluation of MonkeyDB on several applications, showcasing its superior coverage of weak behaviors as well as bug-finding abilities.

Section 4.7 overviews related work, and Section 4.8 concludes.

4.1 Programming Language

Figure 4.1 lists the definition of two simple programming languages that we use to represent applications running on top of Key-Value or SQL stores, respectively. A program is a set of *sessions* running in parallel, each session being composed of a sequence of *transactions*. Each transaction is delimited by `begin` and `commit` instructions[1], and its body contains instructions that access the store, and manipulate a set of local variables Vars ranged over using $x, y, \ldots$.

[1]For simplicity, we assume that all the transactions in the program commit. Aborted transactions can be ignored when reasoning about safety because their effects should be invisible to other transactions.

In case of a program running on top of a Key-Value store, the instructions can be reading the value of a key and storing it to a local variable x (x := read(k)) , writing the value of a local variable x to a key (write(k; x)), or an assignment to a local variable x. The set of values of keys or local variables is denoted by Vals. Assignments to local variables use expressions interpreted as values whose syntax is left unspecified. Each of these instructions can be guarded by a Boolean condition ($\vec{x}$) over a set of local variables $\vec{x}$ (their syntax is not important). Other constructs like while loops can be defined in a similar way. Let P_{KV} denote the set of programs where a transaction body can contain only such instructions.

For programs running on top of SQL stores, the instructions include simplified versions of standard SQL instructions and assignments to local variables. These programs run in the context of a *database schema* which is a (partial) function $S : \mathsf{T} * 2^{\mathsf{C}}$ mapping table names in T to sets of column names in C. The SQL store is an *instance* of a database schema S, i.e., a function D : $\mathsf{dom}(S)$! 2^{R} mapping each table tab in the domain of S to a set of *rows* of tab, i.e., functions $r : S(tab)$! Vals. We use R to denote the set of all rows. The SELECT instruction retrieves the columns $\vec{c}_1$ from the set of rows of tab that satisfy ($\vec{c}_2$) ($\vec{c}_2$ denotes the set of columns used in this Boolean expression), and stores them into a variable x. INSERT adds a new row to tab with values $\vec{x}$, and DELETE deletes all rows from tab that satisfy a condition ($\vec{c}$). The UPDATE instruction assigns the columns $\vec{c}_1$ of all rows of tab that satisfy ($\vec{c}_2$) with values in $\vec{x}$. Let P_{SQL} denote the set of programs where a transaction body can contain only such instructions.

4.2 Operational Semantics for P_{KV}

We define a small-step operational semantics for Key-Value store programs, which is parametrized by a consistency model I. Transactions are executed *serially* one after another, and the values returned by read operations are decided using the axiomatic definition of I. The semantics maintains a history of previously executed operations, and the value returned by a read is chosen non-deterministically as long as extending the current history with the corresponding write-read dependency satisfies the axioms of I. We show that this semantics is sound and complete for any natural consistency model I, i.e., it generates precisely the same set of histories as a *baseline* semantics where transactions can interleave arbitrarily and the read operations can return arbitrary values as long as they can be proved to be correct at the end of the execution.

4.2.1 Definition of the Operational Semantics

We use the program in Figure 4.2a to give an overview of our semantics, assuming Causal Consistency. This program has two concurrent transactions whose reads can both return the initial value 0, which is not possible under Serializability.

Our semantics executes transactions in their entirety one after another (without interleaving them), maintaining a history that contains all the executed operations. We assume that the transaction on the left executes first. Initially, the history contains a fictitious transaction log that writes the initial value 0 to all keys, and that will precede all the transaction logs created during the execution in session order.

Executing a write instruction consists in simply appending the corresponding write operation to the log of the current transaction. For instance, executing the first write (and begin) in our

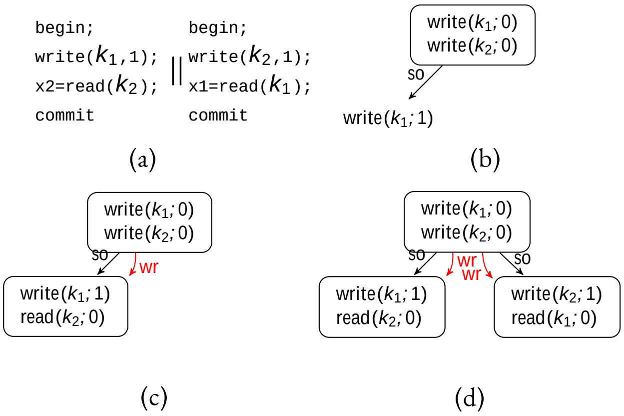


Figure 4.2: The Causal semantics on the program in (a), assuming that the transaction on the left is scheduled first.

example results in adding a transaction log that contains a write operation (see Figure 4.2b). The execution continues with the read instruction from the same transaction, and it cannot switch to the other transaction.

The execution of a read instruction consists in choosing non-deterministically a write-read dependency that validates Causal when added to the current history. In our example, executing `read`(k_2) results in adding a write-read dependency from the transaction log writing initial values, which determines the return value of the `read` (see Figure 4.2c). This choice makes the obtained history satisfy Causal.

The second transaction executes in a similar manner. When executing its read instruction, the chosen write-read dependency is again related to the transaction log writing initial values (see Figure 4.2d). This choice is valid under Causal. Since a read must not read from the preceding transaction, this semantics is able to simulate all the "anomalies" of a weak consistency model (this execution being an example).

Formally, the operational semantics is defined as a transition relation $\Rightarrow_I$ between *configurations*, which are defined as tuples containing the following:

- history h storing the operations executed in the past,
- identifier j of the current session,
- local variable valuation for the current transaction,
- code B that remains to be executed from the current transaction, and
- sessions/transactions P that remain to be executed from the original program.

For readability, we define a program as a partial function $\mathsf{P} : \mathsf{SessId} \rightharpoonup \mathsf{Sess}$ that associates session identifiers in SessId with concrete code as defined in Figure 4.1 (i.e., sequences of transactions). Similarly, the session order so in a history is defined as a partial function $\mathsf{so} : \mathsf{SessId} \rightharpoonup \mathsf{Tlogs}$ that associates session identifiers with sequences of transaction logs. Two transaction logs are ordered by so if one occurs before the other in some sequence $\mathsf{so}(j)$ with $j \in \mathsf{SessId}$.

Before presenting the definition of $\Rightarrow_I$, we introduce some notation. Let h be a history that contains a representation of so as above. We use $h \oplus_j \langle t, O, \mathsf{po} \rangle$ to denote a history where

SPAWN

$$\frac{t \text{ fresh} \quad P(j) = \texttt{begin}; \mathsf{Body}; \texttt{commit}; S}{h, \_, \_, \emptyset, P \Rightarrow_I h \oplus_j \langle t, \emptyset, \emptyset \rangle, j, \emptyset, \emptyset, \mathsf{Body}, P[j \mapsto S]}$$

IF-TRUE

$$\frac{\varphi(\vec{x})[x \mapsto \gamma(x) : x \in \vec{x}] \text{ true}}{h, j, \gamma, \texttt{if}(\varphi(\vec{x}))\{\mathsf{Instr}\}; B, P \Rightarrow_I h, j, \gamma, \mathsf{Instr}; B, P}$$

IF-FALSE

$$\frac{\varphi(\vec{x})[x \mapsto \gamma(x) : x \in \vec{x}] \text{ false}}{h, j, \gamma, \texttt{if}(\varphi(\vec{x}))\{\mathsf{Instr}\}; B, P \Rightarrow_I h, j, \gamma, B, P}$$

WRITE

$$\frac{v = \gamma(x) \quad i \text{ fresh}}{h, j, \gamma, \texttt{write}(k, x); B, P \Rightarrow_I h \oplus_j \mathsf{write}_i(k, v), j, \gamma, B, P}$$

READ-LOCAL

$$\frac{\mathsf{write}(k, v) \text{ is the last write on } k \text{ in } t \text{ w.r.t. } \mathsf{po} \quad i \text{ fresh}}{h, j, \gamma, x := \texttt{read}(k); B, P \Rightarrow_I h \oplus_j \mathsf{read}_i(k, v), j, \gamma[x \mapsto v], B, P}$$

READ-EXTERN

$$\frac{\begin{array}{c} h = (T, \mathsf{so}, \mathsf{wr}) \\ t \text{ is the id of the last transaction log in } \mathsf{so}(j) \quad \mathsf{write}(k, v) \in \mathsf{writes}(t') \text{ with } t' \in T \text{ and } t' \neq t \\ i \text{ fresh} \quad h' = (h \oplus_j \mathsf{read}_i(k, v)) \oplus \mathsf{wr}(t', \mathsf{read}_i(k, v)) \quad h' \text{ satisfies } I \end{array}}{h, j, \gamma, x := \texttt{read}(k); B, P \Rightarrow_I h', j, \gamma[x \mapsto v], B, P}$$

Figure 4.3: Operational semantics for P_{KV} programs under consistency model I. For a function $f : A \rightharpoonup B$, $f[a \mapsto b]$ denotes the function $f' : A \rightharpoonup B$ defined by $f'(c) = f(c)$, for every $c \neq a$ in the domain of f, and $f'(a) = b$.

$\langle t, O, \mathsf{po} \rangle$ is appended to $\mathsf{so}(j)$. Also, for an operation o, $h \oplus_j o$ is the history obtained from h by adding o to the last transaction log in $\mathsf{so}(j)$ and as a last operation in the program order of this log (i.e., if $\mathsf{so}(j) = \sigma; \langle t, O, \mathsf{po} \rangle$, then the session order so' of $h \oplus_j o$ is defined by $\mathsf{so}'(k) = \mathsf{so}(k)$ for all $k \neq j$ and $\mathsf{so}(j) = \sigma; \langle t, O \cup o, \mathsf{po} \cup \{(o', o) : o' \in O\} \rangle$). Finally, for a history $h = \langle T, \mathsf{so}, \mathsf{wr} \rangle$, $h \oplus \mathsf{wr}(t, o)$ is the history obtained from h by adding (t, o) to the write-read relation.

Figure 4.3 lists the rules defining $\Rightarrow_I$. The SPAWN rule starts a new transaction, provided that there is no other live transaction ($B = \emptyset$). It adds an empty transaction log to the history and schedules the body of the transaction. IF-TRUE and IF-FALSE check the truth value of a Boolean condition of an `if` conditional. WRITE corresponds to a write instruction and consists in simply adding a write operation to the current history. READ-LOCAL and READ-EXTERN concern read instructions. READ-LOCAL handles the case where the read follows a write on the same key k in the same transaction: the read returns the value written by the last write on k in the current transaction. Otherwise, READ-EXTERN corresponds to reading a value written in another transaction t' (t is the id of the log of the current transaction). The transaction t' is chosen non-deterministically

$$\textsc{spawn*}\quad \frac{t \text{ fresh} \quad \mathsf{P}(j) = \mathtt{begin}; \mathsf{Body}; \mathtt{commit}; \mathsf{S} \quad \mathsf{B}(j) = \epsilon}{h, \vec{\gamma}, \mathsf{B}, \mathsf{P} \Rightarrow h \oplus_j \langle t, \emptyset, \emptyset \rangle, \vec{\gamma}[j \mapsto \emptyset], \mathsf{B}[j \mapsto \mathsf{Body}], \mathsf{P}[j \mapsto \mathsf{S}]}$$

$$\textsc{read-extern*}\quad \frac{\begin{array}{c}\mathsf{B}(j) = x := \mathtt{read}(k); \mathsf{B} \quad h = (T, \mathsf{so}, \mathsf{wr}) \quad t \text{ is the id of the last transaction log in } \mathsf{so}(j) \\ \mathtt{write}(k, v) \in \mathsf{writes}(t') \text{ with } t' \in \mathsf{compTrans}(h, \mathsf{B}) \text{ and } t \neq t' \\ i \text{ fresh} \quad h' = (h \oplus_j \mathsf{read}_i(k, v)) \oplus \mathsf{wr}(t', \mathsf{read}_i(k, v))\end{array}}{h, \vec{\gamma}, \mathsf{B}, \mathsf{P} \Rightarrow h', \vec{\gamma}[(j, x) \mapsto v], \mathsf{B}[j \mapsto \mathsf{B}], \mathsf{P}}$$

Figure 4.4: A baseline operational semantics for P_{KV} programs. Above, $\mathsf{compTrans}(h, \mathsf{B})$ denotes the set of transaction logs in h that excludes those corresponding to live transactions, i.e., transaction logs $t'' \in T$ such that t'' is the last transaction log in some $\mathsf{so}(j')$ and $B(j') \neq \epsilon$.

as long as extending the current history with the write-read dependency associated to this choice leads to a history that still satisfies I[2].

An *initial* configuration for program P contains the program P along with a history $h = \langle \{t_0\}, \emptyset, \emptyset \rangle$, where t_0 is a transaction log containing only writes that write the initial values of all keys, and empty current transaction code ($\mathsf{B} = \epsilon$). An execution of a program P under an consistency model I is a sequence of configurations $c_0 c_1 \ldots c_n$ where c_0 is an initial configuration for P, and $c_m \Rightarrow_I c_{m+1}$, for every $0 \leq m < n$. We say that c_n is I*-reachable* from c_0. The history of such an execution is the history h in the last configuration c_n. A configuration is called *final* if it contains the empty program ($\mathsf{P} = \emptyset$). Let $\mathsf{hist}_I(\mathsf{P})$ denote the set of all histories of an execution of P under I that ends in a final configuration.

4.2.2 Correctness of the Operational Semantics

We define the correctness of $\Rightarrow_I$ in relation to a *baseline* semantics where transactions can interleave arbitrarily, and the values returned by read operations are only constrained to come from committed transactions. This semantics is represented by a transition relation $\Rightarrow$, which is defined by a set of rules that are analogous to $\Rightarrow_I$. Since it allows transactions to interleave, a configuration contains a history h, the sessions/transactions P that remain to be executed, and:

- a valuation map $\vec{\gamma}$ that records local variable values in the current transaction of each session ($\vec{\gamma}$ associates identifiers of sessions that have live transactions with valuations of local variables),
- a map B that stores the code of each live transaction (associating session identifiers with code).

Figure 4.4 lists some rules defining $\Rightarrow$ (the others can be defined in a similar manner). spawn* starts a new transaction in a session j provided that this session has no live transaction ($\mathsf{B}(j) = \epsilon$). Compared to spawn in Figure 4.3, this rule allows unfinished transactions in other sessions. read-extern* does not check conformance to I, but it allows a read to only return a value written in a completed (committed) transaction. In this work, we consider only consistency models

[2] A history which satisfies the first order formula (3.1) with the axiom defined in figure 3.2 corresponding to I.

satisfying this constraint. Executions, initial and final configurations are defined as in the case of $\Rightarrow_I$. The history of an execution is still defined as the history in the last configuration. Let $\mathsf{hist}(\mathsf{P})$ denote the set of all histories of an execution of P w.r.t. $\Rightarrow$ that ends in a final configuration.

Practical consistency models satisfy a "prefix-closure" property saying that if the axioms of I are satisfied by a pair $\langle h_2, \mathsf{co}_2\rangle$, then they are also satisfied by every *prefix* of $\langle h_2, \mathsf{co}_2\rangle$. A prefix of $\langle h_2, \mathsf{co}_2\rangle$ contains a prefix of the sequence of transactions in h_2 when ordered according to co_2, and the last transaction log in this prefix is possibly incomplete. In general, this prefix-closure property holds for consistency models I that are defined by axioms as in (3.1), provided that the property $\ (t_2, \ )$ is *monotonic*, i.e., the set of models in the context of a pair $\langle h_2, \mathsf{co}_2\rangle$ is a *superset* of the set of models in the context of a prefix $\langle h_1, \mathsf{co}_1\rangle$ of $\langle h_2, \mathsf{co}_2\rangle$. For instance, the property in the axiom defining Causal is $(t_2, \ ) \in (\mathsf{wr} \cup \mathsf{so})^+$, which is clearly monotonic. In general, standard consistency models are defined using a property of the form $(t_2, \ ) \in R$ where R is an expression built from the relations po, so, wr, and co using (reflexive and) transitive closure and composition of relations [17]. Such properties are monotonic in general (they would not be if those expressions would use the negation/complement of a relation). An axiom as in (3.1) is called *monotonic* when the property is monotonic.

The following theorem shows that $\mathsf{hist}_I(\mathsf{P})$ is precisely the set of histories under the baseline semantics, which satisfy I (the validity of the reads is checked at the end of an execution), provided that the axioms of I are monotonic.

Theorem 4.2.1. *For any consistency model I defined by a set of monotonic axioms,* $\mathsf{hist}_I(\mathsf{P}) = \{h \in \mathsf{hist}(\mathsf{P}) : h$ *satisfies* $I\}.$

The direction follows mostly from the fact that $\Rightarrow_I$ is more constrained than $\Rightarrow$. For the opposite direction, given a history h that satisfies I, i.e., there exists a commit order co such that $\langle h, \mathsf{co}\rangle$ satisfies the axioms of I, we can show that there exists an execution under $\Rightarrow_I$ with history h, where transactions execute serially in the order defined by co. The prefix closure property is used to prove that READ-EXTERN transitions are enabled (these transitions get executed with a prefix of h). See the supplementary material for more details.

It can also be shown that $\Rightarrow_I$ is *deadlock-free* for every natural consistency model (e.g., Read Committed, Causal Consistency, Snapshot Isolation, and Serializability), i.e., every read can return some value satisfying the axioms of I at the time when it is executed (independently of previous choices).

4.3 Compiling SQL to Key-Value API

We define an operational semantics for SQL programs (in P_{SQL}) based on a compiler that rewrites SQL queries to Key-Value `read` and `write` instructions. For presentation reasons, we use an intermediate representation where each table of a database instance is represented using a *set* variable that stores values of the primary key[3] (identifying uniquely the rows in the table) and a set of key-value pairs, one for each cell in the table. In a second step, we define a rewriting of the API used to manipulate set variables into Key-Value `read` and `write` instructions.

[3]For simplicity, we assume that primary keys correspond to a single column in the table.

Table:

A		
Id	Name	City
1	Alice	Paris
2	Bob	Bangalore
3	Charles	Bucharest

Intermediate representation:

A = { 1, 2, 3 }

A.1.Id: 1, A.1.Name: Alice, A.1.City: Paris
A.2.Id: 2, A.2.Name: Bob, A.2.City: Bangalore
A.3.Id: 3, A.3.Name: Charles, A.3.City: Bucharest

Figure 4.5: Representing tables with set variables and key-value pairs. We write a key-value pair as key:value.

Intermediate Representation Let $S : \mathsf{T} \rightharpoonup 2^{\mathsf{C}}$ be a database schema (recall that T and C are the set of table names and column names, resp.). For each table tab, let $tab.pkey$ be the name of the primary key column. We represent an instance $D : \mathsf{dom}(S) \to 2^{\mathsf{R}}$ using:

- for each table tab, a set variable tab (with the same name) that contains the primary key value $r(tab.pkey)$ of every row $r \in D(tab)$,
- for each row $r \in D(tab)$ with primary key value $pkeyVal = r(tab.pkey)$, and each column $c \in S(tab)$, a key $tab.pkeyVal.c$ associated with the value $r(c)$.

Example 4.3.1. *The table A on the left of Figure 4.5, where the primary key is defined by the Id column, is represented using a set variable A storing the set of values in the column Id, and one key-value pair for each cell in the table.*

Figure 4.6 lists our rewriting of SQL queries over a database instance D to programs that manipulate the set variables and key-value pairs described above. This rewriting contains the minimal set of accesses to the cells of a table that are needed to implement an SQL query according to its conventional specification. To manipulate set variables, we use `add` and `remove` for adding and removing elements, respectively (returning `true` or `false` when the element is already present or deleted from the set, respectively), and `elements` that returns all of the elements in the input set[4].

SELECT, DELETE, and UPDATE start by reading the contents of the set variable storing primary key values and then, for every row, the columns in e_2 needed to check the Boolean condition (the keys corresponding to these columns). For every row satisfying this Boolean condition, SELECT continues by reading the keys associated to the columns that need to be returned, DELETE removes the primary key value associated to this row from the set tab, and UPDATE writes to the keys corresponding to the columns that need to be updated. In the case of UPDATE, we assume that the values of the variables in $\vec{x}$ are obtained from a valuation (this valuation would be maintained by the operational semantics of the underlying Key-Value store). INSERT adds a new primary key value to the set variable tab (the call to `add` checks whether this value is unique) and then writes to the keys representing columns of this new row.

Manipulating Set Variables Based on the standard representation of a set using its characteristic function, we implement each set variable tab using a set of keys $tab.\mathtt{has}.pkeyVal$, one for

[4] $\mathtt{add}(s, e)$ and $\mathtt{remove}(s, e)$ add and remove the element e from s, respectively. $\mathtt{elements}(s)$ returns the content of s.

SELECT/DELETE/UPDATE

```
rows := elements(tab)
for ( let pkeyVal of rows ) {
   for ( let c of e2 ) {
      val[c] := read(tab.pkeyVal.c)
   if (   [c 7! val[c] : c 2 e2] true )
      // SELECT e1 AS x FROM tab WHERE   (e2)
      for ( let c of e1 )
         out[c] := read(tab.pkeyVal.c)
      x := x [ out
      // DELETE FROM tab WHERE   (e2)
      remove(tab, pkeyVal);
      // UPDATE tab SET e1 = x WHERE   (e2)
      for ( let c of e1 )
         write( tab.pkeyVal.c,   (x[c]) )
```

INSERT INTO *tab* VALUES *x*

```
pkeyVal :=   (x[0])
if ( add(tab,pkeyVal) ) {
   for ( let c of S(tab) ) {
      write( tab.pkeyVal.c,   (x[c]) )
```

Figure 4.6: Compiling SQL queries to the intermediate representation. Above, is a valuation of local variables. Also, in the case of INSERT, we assume that the first element of *x* represents the value of the primary key.

add(*tab*; *pkeyVal*):

```
if (read(tab.has.pkeyVal))
   return false;
write(tab.has.pkeyVal,true)
return true;
```

elements(*tab*):

```
ret := ;
for ( let pkeyVal of Vals )
   if (read(tab.has.pkeyVal))
      ret := ret [ {pkeyVal}
return ret;
```

Figure 4.7: Manipulating set variables using key-value pairs.

each value *pkeyVal* 2 Vals. These keys are associated with Boolean values, indicating whether *pkeyVal* is contained in *tab*. In a concrete implementation, this set of keys need not be fixed a-priori, but can grow during the execution with every new instance of an INSERT. Figure 4.7 lists the implementations of add/elements, which are self-explanatory (remove is analogous).

4.4 Implementation

We implemented MonkeyDB[5] to support an interface common to most storage systems. Operations can be either key-value (KV) updates (to access data as a KV map) or SQL queries (to access data as a relational database). MonkeyDB supports transactions as well; a transaction can include multiple operations. Figure 4.8 shows the architecture of MonkeyDB. A client can connect to

[5] We plan to make MonkeyDB available open-source soon.

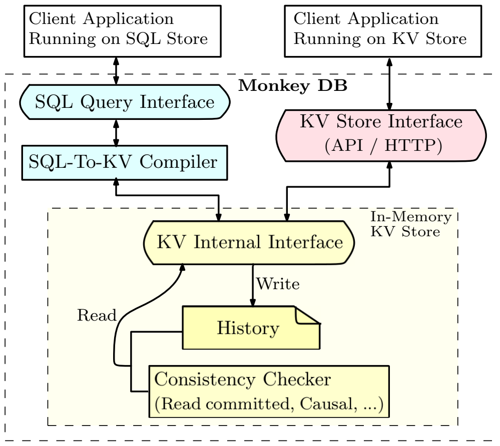


Figure 4.8: Architecture of MonkeyDB

MonkeyDB over a TCP connection, as is standard for SQL databases[6]. This offers a plug-and-play experience when using standard frameworks such as JDBC [83]. Client applications can also use MonkeyDB as a library in order to directly invoke the storage APIs, or interact with it via HTTP requests, with JSON payloads.

MonkeyDB contains a SQL-To-KV compiler that parses an input query[7], builds its Abstract Syntax Tree (AST) and then applies the rewriting steps described in Section 4.3 to produce an equivalent sequence of KV API calls (`read()` and `write()`). It uses a hashing routine (`hash`) to generate unique keys corresponding to each cell in a table. For instance, in order to insert a value *v* for a column *c* in a particular row with primary key value *pkeyVal*, of a table *tab*, we invoke `write(hash(`*tab*`, `*pkeyVal*`, c), v)`. We currently support only a subset of the standard SQL operators. For instance, nested queries or join operators are unsupported; these can be added in the future with more engineering effort.

MonkeyDB schedules transactions from different sessions one after the other using a single global lock. Internally, it maintains execution state as a history consisting of a set of transaction logs, write-read relations and a partial session order (as discussed in §3.1). On a `read()`, MonkeyDB first collects a set of possible writes present in transaction log that can potentially form write-read (read-from) relationships, and then invokes the consistency checker (Figure 4.8) to confirm validity under the chosen consistency model. Finally, it randomly returns one of the values associated with valid writes. A user can optionally instruct MonkeyDB to only select from the set of *latest* valid write per session. This option helps limit weak behaviors for certain reads.

The implementation of our consistency checker is based on prior work [17]. It maintains the write-read relation as a graph, and detects cycles (isolation-level violations) using DFS traversals on the graph. The consistency checker is an independent and pluggable module: we have one for Read Committed and one for Causal Consistency, and more can be added in the future.

[6]We support the MySQL client-server protocol using `https://github.com/jonhoo/msql-srv`.
[7]We use `https://github.com/ballista-compute/sqlparser-rs`

```
// Get user's tweets
Timeline(user u) {
  Begin()
  key = "tweets:" + u.id
  T = read(key)
  Commit()
  return sortByTime(T)
}
```

```
// Get following users' tweets
NewsFeed(user u) {
  Begin()
  FW = read("following:"+ u.id)
  NF = {}
  foreach v ∈ FW:
    T = read("tweets:"+ v.id)
    NF = NF ∪ T
  Commit()
  return sortByTime(NF)
}
```

Figure 4.9: Example operations of the Twitter app

4.5 Evaluation: Microbenchmarks

We consider a set of micro-benchmarks inspired from real-world applications (§4.5.1) and evaluate the number of test iterations required to fail an invalid assertion (§4.5.2). We also measure the *coverage* of weak behaviors provided by MonkeyDB (§4.5.3). Each of these applications were implemented based on their specifications described in prior work; they all use MonkeyDB as a library, via its KV interface.

4.5.1 Applications

Twitter [93] This is based on a social-networking application that allows users to create a new account, follow, unfollow, tweet, browse the newsfeed (tweets from users you follow) and the timeline of any particular user. Figure 4.9 shows the pseudo code for two operations.

A user can access twitter from multiple clients (sessions), which could lead to unexpected behavior under weak consistency models. Consider the following scenario with two users, A and B where user A is accessing twitter from two different sessions, S_1 and S_2. User A views the timeline of user B from one session (S_1:`Timeline(`B`)`) and decides to follow B through another session (S_2:`Follow(`A`, `B`)`). Now when user A visits their timeline or newsfeed (S_2:`NewsFeed(`A`)`), they expect to see all the tweets of B that were visible via `Timeline` in session S_1. But under weak consistency models, this does not always hold true and there could be missing tweets.

Shopping Cart [89] This application allows a user to add, remove and change quantity of items from different sessions. It also allows the user to view all items present in the shopping cart. The pseudo code and an unexpected behavior under weak consistency models were discussed in §1.3, Figure 1.4.

Courseware [77] This is an application for managing students and courses, allowing students to register, de-register and enroll for courses. Courses can also be created or deleted. Courseware maintains the current status of students (registered, de-registered), courses (active, deleted) as well as enrollments. Enrollment can contain only registered students and active courses, subject to the capacity of the course.

Under weak isolation, it is possible that two different students, when trying to enroll concurrently, will both succeed even though only one spot was left in the course. Another example that breaks the application is when a student is trying to register for a course that is being concurrently removed: once the course is removed, no student should be seen as enrolled in that course.

Treiber Stack [76] Treiber stack is a concurrent stack data structure that uses compare-and-swap (CAS) instructions instead of locks for synchronization. This algorithm was ported to operate on a kv-store in prior work [76] and we use that implementation. Essentially, the stack contents are placed in a kv-store, instead of using an in-memory linked data structure. Each row in the store contains a pair consisting of the stack element and the key of the next row down in the stack. A designated key "`head`" stores the key of the top of the stack. CAS is implemented as a transaction, but the `pop` and `push` operations do not use transactions, i.e., each read/write/CAS is its own transaction.

When two different clients try to `pop` from the stack concurrently, under serializability, each `pop` would return a unique value, assuming that each pushed value is unique. However, under causal consistency, concurrent `pop`s can return the same value.

4.5.2 Assertion Checking

We ran the above applications with MonkeyDB to find out if assertions, capturing unexpected behavior, were violated under causal consistency. Table 4.1 summarizes the results. For each application, we used 3 client threads and 3 operations per thread. We ran each test with MonkeyDB for a total of 10,000 times; we refer to a run as an iteration. We report the average number of iterations (Iters) before an assertion failed, and the corresponding time taken (sec). All the assertions were violated within 58 iterations, in half a second or less. In contrast, running with an actual database almost never produces an assertion violation.

Application	Assertion	Avg. time to fail	
		(Iters)	(sec)
Stack	Element popped more than once	3.7	0.02
Courseware	Course registration overflow	10.6	0.09
Courseware	Removed course registration	57.5	0.52
Shopping	Item reappears after deletion	20.2	0.14
Twitter	Missing tweets in feed	6.3	0.03

Table 4.1: Assertions checking results in microbenchmarks

4.5.3 Coverage

The previous section only checked for a particular set of assertions. As an additional measure of test robustness, we count the number of distinct *client-observable states* generated by a test. A client-observable state, for an execution, is the vector of values returned by read operations. For instance, a stack's state is defined by return values of `pop` operations; a shopping cart's state is defined by the return value of `GetCart` and so on.

For this experiment, we randomly generated test harnesses; each harness spawns multiple threads that each execute a sequence of operations. In order to compute the absolute maximum of possible states, we had to limit the size of the tests: either 2 or 3 threads, each choosing between 2 to 4 operations.

Note that any program that concurrently executes operations against a store has two main sources of non-determinism: the first is the interleaving of operations (i.e., the order in which operations are submitted to the store) and second is the choice of read-from (i.e., the value returned by the store under its configured consistency model). MonkeyDB only controls the latter; it is up to the application to control the former. There are many tools that systematically enumerate interleavings (such as Chess [74], Coyote [37]), but we use a simple trick instead to avoid imposing any burden on the application: we included an option in MonkeyDB to deliberately add a small random delay (sleep between 0 to 4 ms) before each transaction begins. This option was sufficient in our experiments, as we show next.

We also implemented a special setup using the Coyote tool [37] to enumerate all sources of non-determinism, interleavings as well as read-from, in order to explore the entire state space of a test. We use this to compute the total number of states. Figure 4.10 shows the number of distinct states observed under different consistency models, averaged across multiple (50) test harnesses. For each of serializability and causal consistency, we show the max (as computed by Coyote) and versions with and without the delay option in MonkeyDB.

Each of these graphs show similar trends: the number of states with causal consistency are much higher than with serializability. Thus, testing with a store that is unable to generate weak behaviors will likely be ineffective. Furthermore, the "delay" versions of MonkeyDB are able to approach the maximum within a few thousand attempts, implying that MonkeyDB's strategy of per-read randomness is effective for providing coverage to the application.

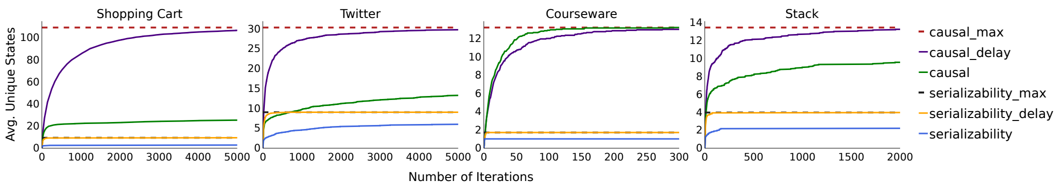


Figure 4.10: State coverage obtained with MonkeyDB for various microbenchmarks

4.6 Evaluation: OLTP Workloads

OLTPBench [39] is a benchmark suite of representative OLTP workloads for relational databases. We picked a subset of OLTPBench for which we had reasonable assertions. Table 4.2 lists basic information such as the number of database tables, the number of static transactions, how many of them are read-only, and the number of different assertions corresponding to system invariants for testing the benchmark. We modified OLTPBench by rewriting SQL join and aggregation operators into equivalent application-level loops, following a similar strategy as prior work [85]. Except for this change, we ran OLTPBench unmodified.

For TPC-C, we obtained a set of 12 invariants from its specification document [92]. For all other benchmarks, we manually identified invariants that the application should satisfy. We asserted these invariants by issuing a read-only transaction to MonkeyDB at the end of the execution of the benchmark. None of the assertions fail under serializability; they are indeed invariants un-

Benchmark	#Tables	#Txns	#Read-only	#Assertions
TPC-C	9	5	2	12
SmallBank	3	6	1	1
Voter	3	1	0	1
Wikipedia	12	5	2	3

Table 4.2: OLTP benchmarks tested with MonkeyDB

der serializability.[8] When using weaker isolation, we configured MonkeyDB to use latest reads only (§4.4) for the assertion-checking transactions in order to isolate the weak behavior to only the application.

We ran each benchmark 100 times and report, for each assertion, the number of runs in which it was violated. Note that OLTPBench runs in two phases. The first is a loading phase that consists of a big initial transaction to populates tables with data, and then the execution phase issues multiple concurrent transactions. With the goal of testing correctness, we *turn down* the scale factor to generate a small load and limit the execution phase time to ten seconds with just two or three sessions. A smaller test setup has the advantage of making debugging easier. With MonkeyDB, there is no need to generate large workloads.

TPC-C TPC-C emulates a wholesale supplier transactional system that delivers orders for a warehouse company. This benchmark deals with customers, payments, orders, warehouses, deliveries, etc. We configured OLTPBench to issue a higher proportion ($> 85\%$) of update transactions, compared to read-only ones. Further, we considered a small input workload constituting of one warehouse, two districts per warehouse and three customers per district.

TPC-C has twelve assertions (A1 to A12) that check for consistency between the database tables. For example, A12 checks: for any customer, the sum of delivered order-line amounts must be equal to the sum of balance amount and YTD (Year-To-Date) payment amount of that customer.

Figure 4.11 shows the percentage of test runs in which an assertion failed. It shows that all the twelve assertions are violated under Read Committed consistency model. In fact, 9 out of the 12 assertions are violated in more than 60% of the test runs. In case of causal, all assertions are violated with three sessions, except for A4 and A11. We manually inspected TPC-C and we believe that both these assertions are valid under causal consistency. For instance, A4 checks for consistency between two tables, both of which are only updated within the same transaction, thus causal consistency is enough to preserve consistency between them.

These results demonstrate the effectiveness of MonkeyDB in breaking (invalid) assertions. Running with MySQL, under read committed, was unable to violate any assertion except for two (A10 and A12), even when increasing the number of sessions to 10. We used the same time limit of 10 seconds for the execution phase. We note that MySQL is much faster than MonkeyDB and ends up processing up to 50 more transactions in the same time limit, yet is unable to violate most assertions. Prior work [85] attempted a more sophisticated test setup where TPC-C was executed on a Cassandra cluster, while running Jepsen [63] for fault injection. This setup also was unable to violate all assertions, even when running without transactions, and on a weaker consistency

[8] We initially observed two assertions failing under serializability. Upon analyzing the code, we identified that the behavior is due to a bug in OLTPBench that we have reported to the authors (link ommitted).

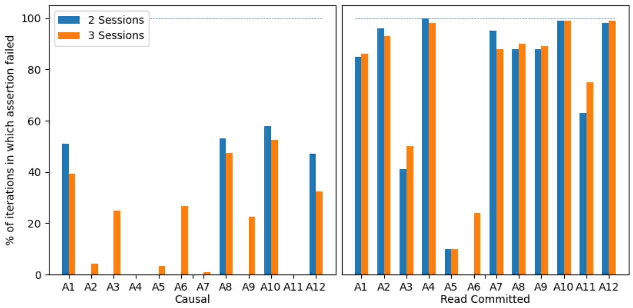


Figure 4.11: Assertion checking: TPC-C

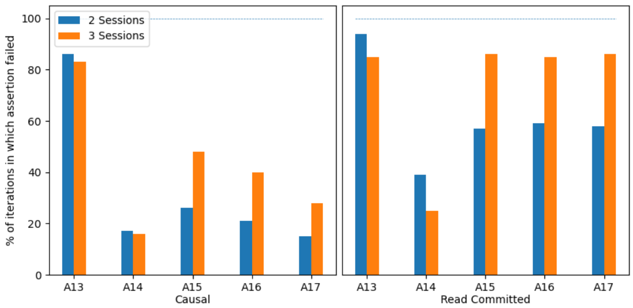


Figure 4.12: Assertion checking: SmallBank, Voter, and Wikipedia

model than read committed. Only six assertions were violated with 10 sessions, eight assertions with 50 sessions, and ten assertions with 100 sessions. With MonkeyDB, there is no need to set up a cluster, use fault injection or generate large workloads that can make debugging very difficult.

SmallBank, Voter, and Wikipedia SmallBank is a standard financial banking system, dealing with customers, saving and checking accounts, money transfers, etc. Voter emulates the voting system of a television show and allows users to vote for their favorite contestants. Wikipedia is based on the popular online encyclopedia. It deals with a complex database schema involving page revisions, page views, user accounts, logging, etc. It allows users to edit its pages and maintains a history of page edits and user actions.

We identified a set of five assertions, A13 to A17, that should be satisfied by these systems. For SmallBank, we check if the total money in the bank remains the same while it is transfered from one account to another (A13). Voter requires that the number of votes by a user is limited to a fixed threshold (A14). For Wikipedia, we check if for a given user and for a given page, the number of edits recorded in the user information, history, and logging tables are consistent (A15-A17). As before, we consider small work loads: (1) five customers for SmallBank, (2) one user for Voter, and (3) two pages and two users for Wikipedia.

Figure 4.12 shows the results. MonkeyDB detected that all the assertions are invalid under the chosen consistency models. Under causal, MonkeyDB could break an assertion in 26.7% (geo-mean) runs given 2 sessions and in 37.2% (geo-mean) runs given 3 sessions. Under read committed, the corresponding numbers are 56.1% and 65.4% for 2 and 3 sessions, respectively.

4.7 Related Work

There have been several directions of work addressing the correctness of database-backed applications. We directly build upon one line of work concerned with the logical formalization of consistency models or isolation levels [4, 12, 17, 31, 97]. Our work relies on the axiomatic definitions of consistency models or isolation levels, as given in [17], which also investigated the problem of checking whether a given history satisfies a certain isolation level. Our kv-store implementation relies on these algorithms to check the validity of the values returned by read operations. Working with a logical formalization allowed us to avoid implementing an actual database with replication or sophisticated synchronization.

Another line of work concentrates on the problem of finding "anomalies": behaviors that are not possible under serializability. This is typically done via a static analysis of the application code that builds a static dependency graph that over-approximates the data dependencies in all possible executions of the application [13, 32, 47, 52, 61, 94]. Anomalies with respect to a given consistency model then corresponds to a particular class of cycles in this graph. Static dependency graphs turn out to be highly imprecise in representing feasible executions, leading to false positives. Another source of false positives is that an anomaly might not be a bug because the application may already be designed to handle the non-serializable behavior [25, 52]. Recent work has tried to address these issues by using more precise logical encodings of the application, e.g. [24, 25] or by using user-guided heuristics [52].

Another approach consists of modeling the application logic and the consistency model in first-order logic and relying on SMT solvers to search for anomalies [62, 75, 78], or defining specialized reductions to assertion checking [10, 11]. The Clotho tool [85], for instance, uses a static analysis of the application to generate test cases with plausible anomalies, which are deployed in a concrete testing environment for generating actual executions.

Our approach, based on testing with MonkeyDB, has several practical advantages. There is no need for analyzing application code; we can work with any application. There are no false positives because we directly run the application and check for user-defined assertions, instead of looking for application-agnostic anomalies. The limitation, of course, is the inherent incompleteness of testing.

Several works have looked at the problem of reasoning about the correctness of applications executing under weak isolation and introducing additional synchronization when necessary [9, 56, 69, 77]. As in the previous case, our work based on testing has the advantage that it can scale to real sized applications (as opposed to these techniques which are based on static analysis or logical proof arguments), but it cannot prove that an application is correct. Moreover, the issue of repairing applications is orthogonal to our work.

From a technical perspective, our operational semantics based on recording past operations and certain data-flow and control-flow dependencies is similar to recent work on stateless model checking in the context of weak memory models, e.g. [1, 66]. This work, however, does not consider transactions. Furthermore, their focus is on avoiding enumerating equivalent executions, which is beyond the scope of our work (but an interesting direction for future work).

4.8 Conclusion

Our goal is to enable developers to test the correctness of their storage-backed applications under weak consistency models. Such bugs are hard to catch because weak behaviors are rarely generated by real storage systems, but failure to address them can lead to loss of business [94]. We present MonkeyDB, an easy-to-use mock storage system for weeding out such bugs. MonkeyDB uses a logical understanding of isolation levels to provide (randomized) coverage of all possible weak behaviors. Our evaluation reveals that using MonkeyDB is very effective at breaking assertions that would otherwise hold under a strong consistency model.

5 Conclusion

In this thesis, we have investigated various algorithmic questions related to automated testing of weakly-consistent data storage systems and applications built on top of them. We have explored the issue of specifying such systems, and studied the theoretical limits of checking whether a given execution satisfies the intended specification. The contributions of this thesis span several directions: (1) new formalisms for specifying weakly-consistent behaviors which integrate data type abstractions like counters, registers, sets, lists, etc, or transactions with various degrees of isolation, (2) new asymptotic complexity results that delineate the tractability of automated testing for data storage systems, and (3) an effective methodology for improving the test coverage of storage-backed applications.

In more detail, Chapter 2 focused on CRDTs, an important class of replicated data types that offers a suitable compromise between consistency and availability. We have introduced a new specification formalism that provides a seamless integration between a particular data type semantics and consistency properties related to the asynchronous propagation of updates. We have used this formalism to derive new complexity results concerning the problem of checking conformance for a given execution.

Chapter 3 investigated the same issues, but in the case of transactional key-value stores. We propose new definitions for established consistency models, which compared to previous approaches, are expressed by logical constraints that follow a common template and make it possible to better distill semantical differences. We have also established interesting semantical relationships between weak consistency models like Prefix Consistency or Snapshot Isolation, and Serializability. These advancements were used to ultimately derive complexity results about checking correctness of transactional key-value store executions, and determine the limits of tractability.

Chapter 4 uses the specification formalism presented in Chapter 3 in order to design a mock in-memory storage system called MonkeyDB that makes it possible to improve coverage in testing applications built on top of transactional storage systems. MonkeyDB simulates the behaviors of a storage system satisfying a specific consistency model by keeping a global history of previously executed operations and making uniform random choices on read operations. Our empirical evaluation shows that MonkeyDB makes it possible to uncover invariant violations in established OLTP benchmarks in a small number of attempts.

5.1 Future Work

The work in this thesis can be advanced along several directions:

- Chapter 2 leaves open several questions related to the complexity of CRDT consistency checking: checking conformance to the counter CRDT when the number of replicas is bounded, or sets and flags when their sizes are also bounded. We believe that these problems remain polynomial time, but as we explained in that chapter, the algorithms introduced in our previous work [16] are only sound.

- Our conformance checking algorithms are *offline*, in the sense, that they receive as input an entire execution. For future work, we want to explore *online* algorithms that process a given execution on the fly. Designing such algorithms with a low resource footprint or small overhead is a highly challenging issue.
- While our algorithms can only be used to indicate whether an execution is correct or not, we would like to investigate the issue of *root-causing* violations. Some bugs are difficult to expose with small length executions. For instance, our tests on AntidoteDB exposed a bug in an execution with 42 transactions, which has been confirmed by the developers, and which cannot be caught with smaller executions (up to our knowledge). In such cases, pin pointing the root cause becomes essential for developers being able to repair it.
- Modern data storage systems support operations/transactions at different levels of consistency. While our work has assumed that all operations/transactions behave under the same consistency model, extending it to such cases is an important research direction.
- Concerning the problem of testing applications, a frequent issue is the lack of precise specifications when checking their correctness against a weak consistency model. An interesting direction for future work is trying to automatically synthesize application-level invariants that distinguish its behaviors under strong consistency versus weak consistency. These invariants could be used during the development process as a way of guiding the insertion of additional synchronization.
- More generally, an important issue is finding the weakest possible consistency model for which an application satisfies the intended specification. This would help in improving the performance of a given application, since weaker consistency models boost concurrency and minimize the synchronization overhead.

Bibliography

1. P. A. Abdulla, S. Aronis, M. F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. "Stateless Model Checking for TSO and PSO". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 353–367. DOI: 10.1007/978-3-662-46681-0_28. URL: https://doi.org/10.1007/978-3-662-46681-0_28.

2. P. A. Abdulla, M. F. Atig, B. Jonsson, M. Lång, T. P. Ngo, and K. Sagonas. "Optimal stateless model checking for reads-from equivalence under sequential consistency". *Proc. ACM Program. Lang.* 3:OOPSLA, 2019, 150:1–150:29. DOI: 10.1145/3360576. URL: https://doi.org/10.1145/3360576.

3. A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Technical report. USA, 1999.

4. A. Adya, B. Liskov, and P. E. O'Neil. "Generalized Isolation Level Definitions". In: *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*. Ed. by D. B. Lomet and G. Weikum. IEEE Computer Society, 2000, pp. 67–78. DOI: 10.1109/ICDE.2000.839388. URL: https://doi.org/10.1109/ICDE.2000.839388.

5. R. Alur, K. L. McMillan, and D. A. Peled. "Model-Checking of Correctness Conditions for Concurrent Objects". *Inf. Comput.* 160:1-2, 2000, pp. 167–188. DOI: 10.1006/inco.1999.2847. URL: https://doi.org/10.1006/inco.1999.2847.

6. Antidote. Retrieved March 28th, 2019. 2019. URL: https://www.antidotedb.eu.

7. Antidote. Retrieved March 28th, 2019. 2019. URL: https://antidotedb.gitbook.io/documentation/overview/configuration.

8. *Azure Cosmos DB Local Emulator*. 2020. URL: https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator.

9. V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. Ed. by L. Réveillère, T. Harris, and M. Herlihy. ACM, 2015, 6:1–6:16. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

10. S. M. Beillahi, A. Bouajjani, and C. Enea. “Checking Robustness Against Snapshot Isolation”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by I. Dillig and S. Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 286–304. DOI: 10.1007/978-3-030-25543-5_17. URL: https://doi.org/10.1007/978-3-030-25543-5_17.

11. S. M. Beillahi, A. Bouajjani, and C. Enea. “Robustness Against Transactional Causal Consistency”. In: *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands*. Ed. by W. J. Fokkink and R. van Glabbeek. Vol. 140. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 30:1–30:18. DOI: 10.4230/LIPIcs.CONCUR.2019.30. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2019.30.

12. H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O’Neil, and P. E. O’Neil. “A Critique of ANSI SQL Isolation Levels”. In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*. Ed. by M. J. Carey and D. A. Schneider. ACM Press, 1995, pp. 1–10. DOI: 10.1145/223784.223785. URL: https://doi.org/10.1145/223784.223785.

13. G. Bernardi and A. Gotsman. “Robustness against Consistency Models with Atomic Visibility”. In: *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. Ed. by J. Desharnais and R. Jagadeesan. Vol. 59. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 7:1–7:15. DOI: 10.4230/LIPIcs.CONCUR.2016.7. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2016.7.

14. J. D. Bingham, A. Condon, and A. J. Hu. “Toward a decidable notion of sequential consistency”. In: *SPAA 2003: Proceedings of the Fifteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 7-9, 2003, San Diego, California, USA (part of FCRC 2003)*. Ed. by A. L. Rosenberg and F. M. auf der Heide. ACM, 2003, pp. 304–313. DOI: 10.1145/777412.777467. URL: https://doi.org/10.1145/777412.777467.

15. J. D. Bingham, A. Condon, A. J. Hu, S. Qadeer, and Z. Zhang. “Automatic Verification of Sequential Consistency for Unbounded Addresses and Data Values”. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*. Ed. by R. Alur and D. A. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer, 2004, pp. 427–439. DOI: 10.1007/978-3-540-27813-9_33. URL: https://doi.org/10.1007/978-3-540-27813-9_33.

16. R. Biswas, M. Emmi, and C. Enea. “On the Complexity of Checking Consistency for Replicated Data Types”. In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*. Ed. by I. Dillig and S. Tasiran. Vol. 11562. Lecture Notes in Computer Science. Springer, 2019, pp. 324–343. DOI: 10.1007/978-3-030-25543-5_19. URL: https://doi.org/10.1007/978-3-030-25543-5_19.

17. R. Biswas and C. Enea. “On the complexity of checking transactional consistency”. *Proc. ACM Program. Lang.* 3:OOPSLA, 2019, 165:1–165:28. DOI: 10.1145/3360591. URL: https://doi.org/10.1145/3360591.

18. R. Biswas and C. Enea. "On the Complexity of Checking Transactional Consistency". *CoRR* abs/1908.04509, 2019. arXiv: 1908.04509. URL: http://arxiv.org/abs/1908.04509.

19. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. "On reducing linearizability to state reachability". *Inf. Comput.* 261, 2018, pp. 383–400. DOI: 10.1016/j.ic.2018.02.014. URL: https://doi.org/10.1016/j.ic.2018.02.014.

20. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. "Verifying Concurrent Programs against Sequential Specifications". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by M. Felleisen and P. Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 290–309. DOI: 10.1007/978-3-642-37036-6_17. URL: https://doi.org/10.1007/978-3-642-37036-6_17.

21. A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. "On verifying causal consistency". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 626–638. URL: http://dl.acm.org/citation.cfm?id=3009888.

22. A. Bouajjani, C. Enea, and J. Hamza. "Verifying eventual consistency of optimistic replication systems". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by S. Jagannathan and P. Sewell. ACM, 2014, pp. 285–296. DOI: 10.1145/2535838.2535877. URL: https://doi.org/10.1145/2535838.2535877.

23. N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. Ed. by A. Birrell and E. G. Sirer. USENIX Association, 2013, pp. 49–60. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

24. L. Brutschy, D. I. Dimitrov, P. Müller, and M. T. Vechev. "Serializability for eventual consistency: criterion, analysis, and applications". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by G. Castagna and A. D. Gordon. ACM, 2017, pp. 458–472. URL: http://dl.acm.org/citation.cfm?id=3009895.

25. L. Brutschy, D. I. Dimitrov, P. Müller, and M. T. Vechev. "Static serializability analysis for causal consistency". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by J. S. Foster and D. Grossman. ACM, 2018, pp. 90–104. DOI: 10.1145/3192366.3192415. URL: https://doi.org/10.1145/3192366.3192415.

26. S. Burckhardt. "Principles of Eventual Consistency". *Found. Trends Program. Lang.* 1:1-2, 2014, pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011.

27. S. Burckhardt, R. Alur, and M. M. K. Martin. "CheckFence: checking consistency of concurrent data types on relaxed memory models". In: *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. Ed. by J. Ferrante and K. S. McKinley. ACM, 2007, pp. 12–21. DOI: 10.1145/1250734.1250737. URL: https://doi.org/10.1145/1250734.1250737.

28. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. "Line-up: a complete and automatic linearizability checker". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by B. G. Zorn and A. Aiken. ACM, 2010, pp. 330–340. DOI: 10.1145/1806596.1806634. URL: https://doi.org/10.1145/1806596.1806634.

29. S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. "Replicated data types: specification, verification, optimality". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by S. Jagannathan and P. Sewell. ACM, 2014, pp. 271–284. DOI: 10.1145/2535838.2535848. URL: https://doi.org/10.1145/2535838.2535848.

30. S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. "Global Sequence Protocol: A Robust Abstraction for Replicated Shared State". In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Ed. by J. T. Boyland. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 568–590. DOI: 10.4230/LIPIcs.ECOOP.2015.568. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2015.568.

31. A. Cerone, G. Bernardi, and A. Gotsman. "A Framework for Transactional Consistency Models with Atomic Visibility". In: *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. Ed. by L. Aceto and D. de Frutos-Escrig. Vol. 42. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 58–71. DOI: 10.4230/LIPIcs.CONCUR.2015.58. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2015.58.

32. A. Cerone and A. Gotsman. "Analysing Snapshot Isolation". *J. ACM* 65:2, 2018, 11:1–11:41. DOI: 10.1145/3152396. URL: https://doi.org/10.1145/3152396.

33. M. Chalupa, K. Chatterjee, A. Pavlogiannis, N. Sinha, and K. Vaidya. "Data-centric dynamic partial order reduction". *Proc. ACM Program. Lang.* 2:POPL, 2018, 31:1–31:30. DOI: 10.1145/3158119. URL: https://doi.org/10.1145/3158119.

34. Cockroach. Retrieved March 28th, 2019. 2019. URL: https://github.com/cockroachdb/cockroach.

35. Cockroach. Retrieved March 28th, 2019. 2019. URL: https://www.cockroachlabs.com/docs/v2.1/transactions.html#isolation-levels.

36. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. "Verifying Correctness of Transactional Memories". In: *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*. IEEE Computer Society, 2007, pp. 37–44. DOI: 10.1109/FAMCAD.2007.40. URL: https://doi.org/10.1109/FAMCAD.2007.40.

37. M. Coyote. *Fearless coding for reliable asynchronous software*. 2019. URL: https://github.com/microsoft/coyote.

38. 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 the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. Ed. by T. C. Bressoud and M. F. Kaashoek. ACM, 2007, pp. 205–220. DOI: 10.1145/1294261.1294281. URL: https://doi.org/10.1145/1294261.1294281.

39. D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases". *Proc. VLDB Endow.* 7:4, 2013, pp. 277–288. DOI: 10.14778/2732240.2732246. URL: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf.

40. N. Eén and N. Sörensson. "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by E. Giunchiglia and A. Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37. URL: https://doi.org/10.1007/978-3-540-24605-3_37.

41. M. Emmi and C. Enea. "Monitoring Weak Consistency". In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. Ed. by H. Chockler and G. Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 487–506. DOI: 10.1007/978-3-319-96145-3_26. URL: https://doi.org/10.1007/978-3-319-96145-3_26.

42. M. Emmi and C. Enea. "Sound, complete, and tractable linearizability monitoring for concurrent collections". *Proc. ACM Program. Lang.* 2:POPL, 2018, 25:1–25:27. DOI: 10.1145/3158113. URL: https://doi.org/10.1145/3158113.

43. M. Emmi and C. Enea. "Weak-consistency specification via visibility relaxation". *Proc. ACM Program. Lang.* 3:POPL, 2019, 60:1–60:28. DOI: 10.1145/3290373. URL: https://doi.org/10.1145/3290373.

44. M. Emmi, C. Enea, and J. Hamza. "Monitoring refinement via symbolic reasoning". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by D. Grove and S. Blackburn. ACM, 2015, pp. 260–269. DOI: 10.1145/2737924.2737983. URL: https://doi.org/10.1145/2737924.2737983.

45. M. Emmi, R. Majumdar, and R. Manevich. "Parameterized verification of transactional memories". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. Ed. by B. G. Zorn and A. Aiken. ACM, 2010, pp. 134–145. DOI: 10.1145/1806596.1806613. URL: https://doi.org/10.1145/1806596.1806613.

46. A. Farzan and P. Madhusudan. "Monitoring Atomicity in Concurrent Programs". In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*. Ed. by A. Gupta and S. Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, 2008, pp. 52–65. DOI: 10.1007/978-3-540-70545-1_8. URL: https://doi.org/10.1007/978-3-540-70545-1_8.
47. A. D. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha. "Making snapshot isolation serializable". *ACM Trans. Database Syst.* 30:2, 2005, pp. 492–528. DOI: 10.1145/1071610.1071615. URL: https://doi.org/10.1145/1071610.1071615.
48. C. Flanagan and P. Godefroid. "Dynamic partial-order reduction for model checking software". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by J. Palsberg and M. Abadi. ACM, 2005, pp. 110–121. DOI: 10.1145/1040305.1040315. URL: https://doi.org/10.1145/1040305.1040315.
49. Galera. Retrieved March 28th, 2019. 2019. URL: http://galeracluster.com.
50. Galera. Retrieved March 28th, 2019. 2019. URL: http://galeracluster.com/documentation-webpages/faq.html.
51. Galera. Retrieved March 28th, 2019. 2019. URL: http://galeracluster.com/documentation-webpages/isolationlevels.html#intra-node-vs-inter-node-isolation-in-galera-cluster.
52. Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang. "IsoDiff: Debugging Anomalies Caused by Weak Isolation". *Proc. VLDB Endow.* 13:11, 2020, pp. 2773–2786. URL: http://www.vldb.org/pvldb/vol13/p2773-gan.pdf.
53. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.
54. P. B. Gibbons and E. Korach. "Testing Shared Memories". *SIAM J. Comput.* 26:4, 1997, pp. 1208–1244. DOI: 10.1137/S0097539794279614. URL: https://doi.org/10.1137/S0097539794279614.
55. S. Gilbert and N. A. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". *SIGACT News* 33:2, 2002, pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.
56. A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodík and R. Majumdar. ACM, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625. URL: https://doi.org/10.1145/2837614.2837625.
57. R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. "Model checking transactional memories". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by R. Gupta and S. P. Amarasinghe. ACM, 2008, pp. 372–382. DOI: 10.1145/1375581.1375626. URL: https://doi.org/10.1145/1375581.1375626.

58. J. Hamza. "On the Complexity of Linearizability". In: *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. Ed. by A. Bouajjani and H. Fauconnier. Vol. 9466. Lecture Notes in Computer Science. Springer, 2015, pp. 308–321. DOI: 10.1007/978-3-319-26850-7_21. URL: https://doi.org/10.1007/978-3-319-26850-7_21.

59. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. "Verifying Sequential Consistency on Shared-Memory Multiprocessor Systems". In: *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*. Ed. by N. Halbwachs and D. A. Peled. Vol. 1633. Lecture Notes in Computer Science. Springer, 1999, pp. 301–315. DOI: 10.1007/3-540-48683-6_27. URL: https://doi.org/10.1007/3-540-48683-6_27.

60. M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Trans. Program. Lang. Syst.* 12:3, 1990, pp. 463–492. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972.

61. S. Jorwekar, A. D. Fekete, K. Ramamritham, and S. Sudarshan. "Automating the Detection of Snapshot Isolation Anomalies". In: *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*. Ed. by C. Koch, J. Gehrke, M. N. Garofalakis, D. Srivastava, K. Aberer, A. Deshpande, D. Florescu, C. Y. Chan, V. Ganti, C. Kanne, W. Klas, and E. J. Neuhold. ACM, 2007, pp. 1263–1274. URL: http://www.vldb.org/conf/2007/papers/industrial/p1263-jorwekar.pdf.

62. G. Kaki, K. Earanky, K. C. Sivaramakrishnan, and S. Jagannathan. "Safe replication through bounded concurrency verification". *Proc. ACM Program. Lang.* 2:OOPSLA, 2018, 164:1–164:27. DOI: 10.1145/3276534. URL: https://doi.org/10.1145/3276534.

63. K. Kingsbury. Retrieved March 28th, 2019. 2013-2019. URL: http://jepsen.io.

64. K. Kingsbury. Retrieved March 28th, 2019. 2015. URL: https://github.com/jepsen-io/jepsen/blob/master/galera/src/jepsen/galera/dirty_reads.clj.

65. K. Kingsbury. *Jepsen: Distributed Systems Safety Research*. 2016. URL: https://jepsen.io.

66. M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. "Effective stateless model checking for C/C++ concurrency". *Proc. ACM Program. Lang.* 2:POPL, 2018, 17:1–17:32. DOI: 10.1145/3158105. URL: https://doi.org/10.1145/3158105.

67. L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs". *IEEE Trans. Computers* 28:9, 1979, pp. 690–691. DOI: 10.1109/TC.1979.1675439. URL: https://doi.org/10.1109/TC.1979.1675439.

68. L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Commun. ACM* 21:7, 1978, pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

69. C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by G. Gibson and N. Zeldovich. USENIX Association, 2014, pp. 281–292. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li_cheng_2.

70. G. Lowe. "Testing for linearizability". *Concurr. Comput. Pract. Exp.* 29:4, 2017. DOI: 10.1002/cpe.3928. URL: https://doi.org/10.1002/cpe.3928.

71. H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. "Existential consistency: measuring and understanding consistency at Facebook". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by E. L. Miller and S. Hand. ACM, 2015, pp. 295–310. DOI: 10.1145/2815400.2815426. URL: https://doi.org/10.1145/2815400.2815426.

72. J. Manson, W. Pugh, and S. V. Adve. "The Java memory model". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by J. Palsberg and M. Abadi. ACM, 2005, pp. 378–391. DOI: 10.1145/1040305.1040336. URL: https://doi.org/10.1145/1040305.1040336.

73. A. W. Mazurkiewicz. "Trace Theory". In: *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Vol. 255. Lecture Notes in Computer Science. Springer, 1986, pp. 279–324. DOI: 10.1007/3-540-17906-2_30. URL: https://doi.org/10.1007/3-540-17906-2_30.

74. M. Musuvathi and S. Qadeer. "Fair stateless model checking". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by R. Gupta and S. P. Amarasinghe. ACM, 2008, pp. 362–371. DOI: 10.1145/1375581.1375625. URL: https://doi.org/10.1145/1375581.1375625.

75. K. Nagar and S. Jagannathan. "Automated Detection of Serializability Violations Under Weak Consistency". In: *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. Ed. by S. Schewe and L. Zhang. Vol. 118. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 41:1–41:18. DOI: 10.4230/LIPIcs.CONCUR.2018.41. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2018.41.

76. K. Nagar, P. Mukherjee, and S. Jagannathan. "Semantics, Specification, and Bounded Verification of Concurrent Libraries in Replicated Systems". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by S. K. Lahiri and C. Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 251–274. DOI: 10.1007/978-3-030-53288-8_13. URL: https://doi.org/10.1007/978-3-030-53288-8_13.

77. S. S. Nair, G. Petri, and M. Shapiro. "Proving the Safety of Highly-Available Distributed Objects". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by P. Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 544–571. DOI: 10.1007/978-3-030-44914-8_20. URL: https://doi.org/10.1007/978-3-030-44914-8_20.

78. B. Ozkan. "Verifying Weakly Consistent Transactional Programs using Symbolic Execution". In: *In Proc. 8th Int. Conference on Networked Systems (NETYS)*. 2020.

79. B. K. Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher. “Randomized testing of distributed systems with probabilistic guarantees”. *Proc. ACM Program. Lang.* 2:OOPSLA, 2018, 160:1–160:28. DOI: 10.1145/3276530. URL: https://doi.org/10.1145/3276530.

80. C. H. Papadimitriou. “The serializability of concurrent database updates”. *J. ACM* 26:4, 1979, pp. 631–653. DOI: 10.1145/322154.322158. URL: https://doi.org/10.1145/322154.322158.

81. A. Pavlo. “What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research”. In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu. ACM, 2017, p. 3. DOI: 10.1145/3035918.3056096. URL: https://doi.org/10.1145/3035918.3056096.

82. J. R. G. Paz. *Microsoft Azure Cosmos DB Revealed: A Multi-Modal Database Designed for the Cloud*. 1st. Apress, USA, 2018. ISBN: 1484233506.

83. J. Platform. *JDBC: Java Database Connectivity API*. URL: https://en.wikipedia.org/wiki/Java_Database_Connectivity.

84. S. Qadeer. “Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking”. *IEEE Trans. Parallel Distributed Syst.* 14:8, 2003, pp. 730–741. DOI: 10.1109/TPDS.2003.1225053. URL: https://doi.org/10.1109/TPDS.2003.1225053.

85. K. Rahmani, K. Nagar, B. Delaware, and S. Jagannathan. “CLOTHO: directed test generation for weakly consistent database systems”. *Proc. ACM Program. Lang.* 3:OOPSLA, 2019, 117:1–117:28. DOI: 10.1145/3360543. URL: https://doi.org/10.1145/3360543.

86. H. Roh, M. Jeon, J. Kim, and J. Lee. “Replicated abstract data types: Building blocks for collaborative applications”. *J. Parallel Distributed Comput.* 71:3, 2011, pp. 354–368. DOI: 10.1016/j.jpdc.2010.12.006. URL: https://doi.org/10.1016/j.jpdc.2010.12.006.

87. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. “Conflict-Free Replicated Data Types”. In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Ed. by X. Défago, F. Petit, and V. Villain. Vol. 6976. Lecture Notes in Computer Science. Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3_29.

88. M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. “Convergent and Commutative Replicated Data Types”. *Bull. EATCS* 104, 2011, pp. 67–88. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/120.

89. K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan. “Declarative programming over eventually consistent data stores”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by D. Grove and S. Blackburn. ACM, 2015, pp. 413–424. DOI: 10.1145/2737924.2737981. URL: https://doi.org/10.1145/2737924.2737981.

90. D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994*. IEEE Computer Society, 1994, pp. 140–149. DOI: 10.1109/PDIS.1994.331722. URL: https://doi.org/10.1109/PDIS.1994.331722.

91. D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by M. B. Jones. ACM, 1995, pp. 172–183. DOI: 10.1145/224056.224070. URL: https://doi.org/10.1145/224056.224070.

92. *TPC-C Specification*. 2020. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

93. Twissandra. *Twitter clone on Cassandra*. Accessed November 1, 2020. URL: https://github.com/twissandra/twissandra.

94. T. Warszawski and P. Bailis. "ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu. ACM, 2017, pp. 5–20. DOI: 10.1145/3035918.3064037. URL: https://doi.org/10.1145/3035918.3064037.

95. J. M. Wing and C. Gong. "Testing and Verifying Concurrent Objects". *J. Parallel Distributed Comput.* 17:1-2, 1993, pp. 164–182. DOI: 10.1006/jpdc.1993.1015. URL: https://doi.org/10.1006/jpdc.1993.1015.

96. P. Wolper. "Expressing Interesting Properties of Programs in Propositional Temporal Logic". In: *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 1986, pp. 184–193. DOI: 10.1145/512644.512661. URL: https://doi.org/10.1145/512644.512661.

97. A. X3. *135-1992. American National Standard for Information Systems-Database Language-SQL*. Technical report. 1992.