Reasoning About Replicated Data Types

Constantin Enea
IRIF, University of Paris
Geo-Distributed Software Systems

- large numbers of users
- distributed across wide geographical areas
- generate and access huge amounts of data

Examples:
- online markets (Amazon)
- government services (tax payment)
- enterprise customer services
- social networks (Facebook)
- managing connected devices and sensors (Internet of Things)
Distributed Data Storage - Replicated Data Types

- to support failures, data is **replicated**
- interface restricted to a **fixed set of methods** (defining a data type)
  - key-value stores (shared memories) provide methods `write(key,value)` and `read(key)`
- restricted interface: **different approach** than relational databases (better scalability)
Replication vs. Consistency

• **conflicting concurrent** updates => how they are observed on different replicas ?

• adversarial environments: crashes, network partitions
Pessimistic Replication

- using consensus algorithms to agree on an order between conflicting concurrent updates
  - **strong consistency** ✓
  - **availability** ❌
- CAP theorem [Gilbert et al.’02]: guaranteeing strong consistency + availability + partition tolerance is impossible
Optimistic Replication

- each update is applied on the local replica and propagated in the background
- replicas may store different versions of data: weak consistency

• strong consistency
  - availability
Consistency Criteria

Strong Consistency

- **sequential consistency** [Lamport’79]: interleaving semantics
- **linearizability** [Herlihy&Wing’90]: interleavings must be consistent with real-time

Weak Consistency

- **eventual consistency** [Terry et al.’95]: reads are safe + eventual propagation of updates and conflict resolution
- **session guarantees** [Terry et al.’94]: consistency w.r.t. previous operations in the same session
- **causal consistency** [Lamport’78]: causally-related operations are observed in the same order

(many other variations, including quantitative versions:

Viotti, Vukolic: *Consistency in Non-Transactional Distributed Storage Systems*)
Things can go wrong: databases

"Over the past six years, Jepsen has analyzed over two dozen databases, coordination services, and queues—and we’ve found replica divergence, data loss, stale reads, read skew, lock conflicts, and much more."

https://jepsen.io
Weak Consistency: Anomalies

**Positive Counter** Data Type (representing for instance a bank account)

- dealing with concurrent withdrawal operations
- replicas don’t synchronize after every withdrawal
- the sum of the withdrawals may be greater than the current value

![Diagram showing two concurrent withdrawals from a bank account with a positive counter, resulting in a balance greater than the total withdrawal amount.](image)
Things can go wrong: applications

security vulnerabilities due to weak consistency

- Flexcoin Bitcoin exchange
  (numerous requests => weak consistency => overdrawn accounts)
- vulnerabilities in applications that back 2M+ eCommerce sites

ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications

Todd Warszawski, Peter Bailis
Stanford InfoLab

ABSTRACT
In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions. While low transaction volumes mask many potential concurrency-related errors under normal operation, determined adversaries can exploit these programmatically for fun and profit. In this paper, we formalize a new kind of attack on database-backed applications called an ACIDRain attack, in which an adversary systematically exploits concurrency-related vulnerabilities via programmatically accessible APIs. These attacks are not theoretical: ACIDRain attacks have already occurred in a handful of applications in the wild, including one attack which bankrupted a popular Bitcoin exchange. To proactively detect the potential for ACIDRain attacks, we extend the theory of weak isolation to analyze latent potential for non-serializable behavior under concurrent web API calls. We introduce a language-agnostic method for detecting potential isolation anomalies in web applications, called Abstract Anomaly Detection (2AD), that uses dynamic traces of database accesses to efficiently reason about the space of possible concurrent turnstiles. We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in 10 languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, overspend gift cards, and steal inventory.

1. INTRODUCTION
For decades, database systems have been tasked with maintaining application integrity despite concurrent access to shared state [39]. The serializable transaction concept dictates that, if programmers configure the database isolation level but often default to non-serializable levels [17, 19] that may corrupt application state [45]. Moreover, we are unaware of any systematic study that examines whether programmers correctly utilize transactions.

Figure 1: (a) A simplified example of code that is vulnerable to an ACIDRain attack allowing overdraft under concurrent access. Two concurrent instances of the withdraw function could both read balance $100, check that $100 > $99, and each allow $99 to be withdrawn, resulting in $198 total withdrawals. (b) Example of how transactions could be inserted to address this error. However, even this code is vulnerable to attack at isolation levels at or below Read Committed, unless explicit locking such as SELECT FOR UPDATE is used. While this scenario closely resembles textbook examples of improper transaction use, in this paper we show that widely-deployed eCommerce applications are similarly vulnerable to such ACIDRain attacks, allowing corruption of application state and theft of assets.
Challenges

Specifications expressed in a formal (mathematical) language
- expressing developers' intent and user expectations

Automated testing
- generating interesting clients
- efficient algorithms for per-execution verification

Static verification
- (interactive) algorithms for verifying all executions of an implementation
- reference implementations

Reasoning about applications
- proving application robust (having the same behaviors as under strong consistency)
- program logics for reasoning on top of weak consistency
Outline

• Formalizing Weak Consistency
• Testing Causal Consistency of Key-Value Stores
• Testing Transactional Databases
Outline

• Formalizing Weak Consistency
• Testing Causal Consistency of Key-Value Stores
• Testing Transactional Databases
Histories

- modeling **client observations** in executions
- history = (set of ops., *session order*)
- **operations**: methods, inputs, outputs
- **session**: sequential interface to the replicated storage (connection)

```
write(x,1)  
|       |       |
|       |       |
read(x) => 2 | read(x) => 2

“1st session”        “2nd session”
```
Abstract Executions

- modeling propagation and conflict resolution in executions
- abs. execution = (history, visibility, arbitration)
  - visibility: binary relation between operations
    - visibility ∪ session order is acyclic
  - arbitration: total order on operations (modeling timestamps)

```
write(x,1) -> write(x,2)  
read(x) => 2  
```

“1st session”  “2nd session”

```
write(x,1)  
write(x,2)  
read(x) => 2  
read(x) => 2  
```
Axiomatic Definitions of Consistency Criteria

- history $\vDash X$ iff $\exists$ visibility, arbitration such that
  
  $$(\text{history, visibility, arbitration}) \vDash \text{Axioms}(X)$$

- The axioms describe:
  - data type semantics (meaning of operations + conflict resolution)
  - ordering guarantees (constraints relating session order, visibility, arbitration)
  - convergence (e.g., replicas eventually agree on an order between updates)
Data Type Semantics

- **operation context** = the set of visible operations, related by **visibility** and **arbitration**

```
write(x,1)  write(x,2)
read(x) => 2  read(x) => 2  read(x) => 2

"1st session"  "2nd session"

op. context( read(x) => 2 ) =
```

```
write(x,1)  write(x,2)
```

Data Type Semantics

• operation context = the set of visible operations, related by visibility and arbitration

“1st session”

write\(x,1\)

\(\text{op. context(} \quad \text{read}(x) \Rightarrow 2 \quad \text{)} = \)

write\(x,2\)

\(\text{read}(x) \Rightarrow 2\)

\(\text{read}(x) \Rightarrow 2\)

“2nd session”

• Data type specification: mapping (invocation, context) to return values

Register Spec. \(\begin{pmatrix} \text{read}(x) \\ \text{write}(x,1) \end{pmatrix} = 2\)

(standard sequential specification)
Data Type Semantics

• operation context = the set of visible operations, related by visibility and arbitration

Data type specification: mapping (invocation, context) to return values

Return Value Axiom on abstract executions e = (history, visibility, arbitration):

\[ \text{RetVal}(\text{Spec}): \forall \, \text{op} \in \text{history}. \, \text{rval}(\text{op}) = \text{Spec}(\text{inv}(\text{op}), \text{context}(e, \text{op})) \]
Non-Sequential Specifications

- conflict resolution $\neq$ timestamps (arbitration)

```
write(x,0)

write(x,1)
read(x) => {2}

write(x,2)
read(x) => {1,2}
write(x,0)
```
Non-Sequential Specifications

• conflict resolution ≠ timestamps (arbitration)
Non-Sequential Specifications

• conflict resolution ≠ timestamps (arbitration)

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
read(x) => {2}
read(x) => {2}
read(x) => {1,2}
read(x) => {1,2}
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
read(x) => {1,2}
```

```
read(x) => {1,2}
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```

```
write(x,0)
write(x,1)
write(x,2)
```
Non-Sequential Specifications

- conflict resolution ≠ timestamps (arbitration)

Multi-Value Register Spec. $\left( \begin{array}{c} \text{read}(x) \\ \text{write}(x,0) \\ \text{write}(x,1) \\ \text{write}(x,2) \end{array} \right) = \{1,2\}$
Consistency Criteria

Ordering Guarantees

- **ReadMyWrites**: 
  \[ \text{so} \subseteq \text{vis} \]

- **MonotonicReads**: 
  \[ \text{vis}; \text{so} \subseteq \text{vis} \]

- **CausalVisibility**: 
  ReadMyWrites + vis transitive

- **CausalArbitration**: 
  \[ \text{vis} \subseteq \text{arb} \]

- **EventualVisibility**: 
  \[ \forall o. \{ o' : (o, o') \notin \text{vis} \} < \infty \]
  (the nb. of operations not seeing o is finite)

Eventual Consistency(Spec) = \text{RVal}(\text{Spec}) \land \text{EventualVisibility}

Causal Consistency(Spec) = \text{RVal}(\text{Spec}) \land \text{CausalVisibility}

Causal Convergence(Spec) = Causal Consistency(Spec) \land \text{CausalArbitration}
Outline

- Formalizing Weak Consistency
- Testing Causal Consistency of Key-Value Stores
- Testing Transactional Databases
Causal consistency

write(x,1)

read(x) => 1

write(x,2)

read(x) => ?

How are they observed in other sessions (replicas)?
Causal consistency

write(x,1) causally related write(x,2) in the causal order

read(x) => 1

read(x) => 1

read(x) => 1

read(x) => 2

read(x) => 2

How are they observed in other sessions (replicas)?
Variations of Causal consistency

How are they observed in other sessions (replicas)?

- **Causedly related**
  - Observed in the causal order
  - Observed in the same order in all sessions: causal convergence
  - Observed in the same order during a session: causal memory

What about causally-unrelated writes?

- Can be observed in **different** orders: causal consistency
- Observed in the **same** order in **all sessions**: causal convergence
- Observed in the **same** order during a session: causal memory
Axiomatization Complexity

- history ⊨ Causal Cons. iff ∃ vis, arb such that
  
  \[(\text{history, vis, arb}) \models \text{RVal}(\text{Register})\]

  \[\land \text{so} \subseteq \text{vis} \land \text{vis transitive}\]

- Register \((\text{read(x)} \Rightarrow \_\lt\text{, context}) = \text{one of the maximal values w.r.t. the visibility relation}\)

- Complexity:
  - second-order quantification over vis
  - specifications expressed in terms of operation contexts
Simplifying the Axiomatization

• only a single write in an operation context is important for justifying a read
  • write-read (wr) relates \texttt{write}(x,a) to \texttt{read}(x) \Rightarrow a
  • visibility \Rightarrow \text{causal order (co)} = ( \text{so} \cup \text{wr} )^+
Simplifying the Axiomatization

• only a single write in an operation context is important for justifying a read
  • write-read (wr) relates write(x,a) to read(x) => a
  • visibility => causal order (co) = ( so u wr )+

• maximality in operation context
  • read-write (rw) relates read(x) => a to write(x,b) that “overwrites” the read value
Simplifying the Axiomatization

• only a single write in an operation context is important for justifying a read
  • write-read (wr) relates \text{write}(x,a) to \text{read}(x) \Rightarrow a
  • visibility => causal order (co) = ( so \cup wr )^+

• maximality in operation context
  • read-write (rw) relates \text{read}(x) \Rightarrow a to \text{write}(x,b) that “overwrites” the read value

\begin{align*}
\text{write}(x,a) & \xrightarrow{\text{co}} \text{write}(x,b) \\
\downarrow \text{wr} & \quad \text{rw} \quad \downarrow \text{wr}
\end{align*}

• history \nvDash Causal Cons. iff \exists w r such that

\[( \text{history, wr } ) \nvDash \text{so } \cup \text{wr } \cup \text{rw } \text{is acyclic}\]
Using the Simplified Axiomatization in Testing

Bouajjani et al: On Verifying Causal Consistency (POPL’17)

\[
\text{history} \models \text{Causal Cons.} \iff \exists \, \text{wr} \text{ such that } \\
( \text{history, wr} ) \models \text{so} \cup \text{wr} \cup \text{rw} \text{ is acyclic}
\]

- key-value store implementations are \textbf{data independent} (their behavior doesn’t depend on the written values)

- for such implementations, it is sound to consider only \textbf{differentiated} executions where each value is written \textbf{at most once}

- the \text{wr} relation in differentiated executions is \textbf{fixed}

- acyclicity of \text{so} \cup \text{wr} \cup \text{rw} can be decided in polynomial time

- similar extensions for causal convergence, causal memory, CRDTs
Outline

- Formalizing Weak Consistency
- Testing Causal Consistency of Key-Value Stores
- Testing Transactional Databases
Transactions simplify concurrent programming

- blocks of instructions executed in isolation and resilient to failures

Modern databases provide different levels of consistency

- serializability carries a significant penalty on availability

\[
\begin{align*}
\text{read}(x,0) & \quad \text{read}(x,1) \\
\text{read}(y,0) & \quad \text{read}(y,0) \\
x = 1 & \quad y = 1 \\
\end{align*}
\]

\[\models\text{serializability}\]

- weak consistency: better tradeoffs between consistency and availability

\[
\begin{align*}
\text{read}(x,0) & \quad \text{read}(x,0) \\
\text{read}(y,0) & \quad \text{read}(y,0) \\
x = 1 & \quad y = 1 \\
\end{align*}
\]

\[\not\models\text{snapshot isolation}\]
Committed transaction = set of reads \( \text{read}(x,v) \) or writes \( x = v \)

- at most **one write per variable** (only the last write is visible)
- **reads precede writes** (a read of \( x \) following \( x = v \) returns value \( v \))

History = a set of (committed) transactions along with a (strict, partial) session order \( \text{so} \) and a write-read relation \( \text{wr} \)

- the inverse of \( \text{wr} \) is total
- \( \text{wr} \cup \text{so} \) is acyclic
Formalizing Weak Consistency

**History** \( h \) satisfies **criterion** \( X \) iff there exists a strict total **commit order** \( co \) such that:

- \( wr \cup so \subseteq co \)
- \( h, co \models \text{Axioms}(X) \)

**Axioms:** for all transactions \( t_1, t_2, t_3 \) and variable \( x \),

\[
(t_1, t_3) \in wr_x \land t_2 \text{ writes } x \land R(t_2, t_3) \Rightarrow (t_2, t_1) \in co
\]
Read Atomic

\[ x = 1 \quad \text{read}(x,1) \quad \text{commit} \]

“every read value is written by a committed transaction”
(Read Committed)
Read Atomic

“every read value is written by a committed transaction” (Read Committed)

“successive reads of the same variable return the same value” (Repeatable Read)
Read Atomic

“every read value is written by a committed transaction” (Read Committed)

“successive reads of the same variable return the same value” (Repeatable Read)

“the read values are written by visible transactions which are maximal w.r.t. commit order”
Read Atomic

x = 1
read(x,1)
abort

Read Committed
“every read value is written by a committed transaction”

x = 1
read(x,1)
x = 2
read(x,2)

Repeatable Read
“successive reads of the same variable return the same value”

x = 1
y = 1
read(x,1)
read(y,2)
x = 2
y = 2

Maximal W.R.T. Commit Order
“the read values are written by visible transactions which are maximal w.r.t. commit order”

writes x

\[ t_2 \rightarrow wr \rightarrow t_3 \]
\[ t_1 \rightarrow wr_x \rightarrow t_3 \]
Read Atomic

- \( x = 1 \), read(\( x, 1 \))
- \( x = 1 \), read(\( x, 2 \))
- \( x = 1 \), read(\( x, 1 \))
- \( x = 1 \), read(\( x, 2 \))

“every read value is written by a committed transaction” (Read Committed)

- \( x = 2 \)
- \( x = 2 \)
- \( x = 2 \)
- \( x = 2 \)

“successive reads of the same variable return the same value” (Repeatable Read)

- \( y = 1 \)
- \( y = 2 \)
- \( y = 2 \)
- \( y = 2 \)

“the read values are written by visible transactions which are maximal w.r.t. commit order”

writes \( x \)

\( t_2 \rightarrow \text{wr} \rightarrow t_3 \)

\( t_1 \rightarrow \text{wr}_x \rightarrow t_3 \)

\( t_1 \rightarrow \text{co} \rightarrow t_2 \)

\( x = 1 \) abort

\( x = 2 \) commit

\( x = 1 \) abort

\( x = 2 \) commit

\( y = 1 \) abort

\( y = 2 \) commit

\( y = 2 \) abort

\( y = 2 \) commit
Read Atomic

- **x = 1**
  - abort
  - commit

  “every read value is written by a committed transaction” (Read Committed)

- **x = 1**
  - read(x,1)

  “successive reads of the same variable return the same value” (Repeatable Read)

- **x = 2**
  - read(x,2)

- **x = 1**
  - read(x,1)

  “the read values are written by visible transactions which are maximal w.r.t. commit order”

writes x

\[ t_2 \xrightarrow{wr} t_3 \]

\[ t_1 \xrightarrow{wr_y} t_3 \]

\[ t_1 \xrightarrow{wr_x} t_3 \]
Read Atomic

\( x = 1 \)
- abort
- commit

“every read value is written by a committed transaction”
(Read Committed)

\( x = 1 \)
- read(x,1)

\( x = 2 \)
- read(x,2)

“successive reads of the same variable return the same value”
(Repeatable Read)

\( x = 1 \)
\( y = 1 \)
- read(x,1)
- read(y,2)

“the read values are written by visible transactions which are maximal w.r.t. commit order”
Read Atomic

\[
\begin{align*}
 x &= 1 & \text{read}(x,1) & \text{x} & \text{abort} \\
 & & \text{commit} & \text{x} & \text{x}
\end{align*}
\]

“every read value is written by a committed transaction” (Read Committed)

\[
\begin{align*}
 x &= 1 & \text{read}(x,1) & \text{x} & \text{read}(x,2) & \text{x} & \text{x}
\end{align*}
\]

“successive reads of the same variable return the same value” (Repeatable Read)

\[
\begin{align*}
 x &= 1 & y &= 1 & \text{read}(x,1) & \text{x} & \text{x}
\end{align*}
\]

“the read values are written by visible transactions which are maximal w.r.t. commit order”

writes x

\[
\begin{align*}
 t_2 & \xrightarrow{wr} t_3 \\
 t_1 & \xrightarrow{wr_x} t_3 \\
 & \xrightarrow{co} t_1
\end{align*}
\]
Read Atomic

\[
x = 1
]\]
abort
\[
x = 2
]\]
commit

“every read value is written by a committed transaction”
(Read Committed)

\[
x = 1
\]
read(x,1)
\[
x = 1
\]
read(x,1)
\[
x = 2
\]
read(x,2)
\[
x = 1
\]
read(x,1)
\[
x = 2
\]
read(x,2)
\[
x = 1
\]
read(x,1)
\[
x = 2
\]
read(y,2)

“successive reads of the same variable return the same value”
(Repeatable Read)

“the read values are written by visible transactions which are maximal w.r.t. commit order”

writes x

\[
t_1 \xrightarrow{wr_x} t_3
\]
\[
t_2 \xrightarrow{wr} t_3
\]
\[
t_1 \xrightarrow{co} t_2
\]
\[
t_2 \xrightarrow{co} t_3
\]
\[
t_1 \xrightarrow{co} t_2
\]
\[
x = 1
\]
\[
x = 1
\]
\[
x = 1
\]
\[
x = 2
\]
\[
x = 2
\]
\[
x = 1
\]
\[
x = 2
\]
\[
x = 2
\]
\[
y = 1
\]
\[
y = 1
\]
\[
y = 2
\]
Tractable Criteria

**Read Atomic**

- \( \text{writes } x \)
- \( t_2 \rightarrow t_1 \rightarrow t_3 \)
- \( \text{co} \rightarrow t_2 \)

**Causal Consistency**

- \( \text{writes } x \)
- \( t_2 \rightarrow (\text{wr} \cup \text{so})^+ \)
- \( \text{co} \rightarrow t_2 \rightarrow t_1 \rightarrow t_3 \)

Checking consistency:

- constructing a partial commit order through saturation (DATALOG)
- absence of cycles => satisfying the criterion
Checking any of these criteria is NP-complete.

Polynomial-time when the number of sessions is fixed.
• enumerating one-transaction extensions of prefixes of the session order
Checking Serializability

- enumerating one-transaction extensions of prefixes of the session order
- each prefix is “serializable”: the history admits a valid commit order which linearizes only transactions in that prefix (the commit order satisfies the Ser. axiom)
Checking Serializability

- enumerating one-transaction extensions of prefixes of the session order

- each prefix is “serializable”: the history admits a valid commit order which linearizes only transactions in that prefix (the commit order satisfies the Ser. axiom)

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- enumerating one-transaction extensions of prefixes of the session order
- each prefix is "serializable": the history admits a valid commit order which linearizes only transactions in that prefix (the commit order satisfies the Ser. axiom)
- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- enumerating one-transaction extensions of prefixes of the session order

- each prefix is “serializable”: the history admits a valid commit order which linearizes only transactions in that prefix (the commit order satisfies the Ser. axiom)

- check that each extension leads to a serializable prefix; if not, backtrack

- every prefix is represented precisely by its maximal elements (an antichain of session order)
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Checking Serializability

- check that each extension leads to a serializable prefix; if not, backtrack
Reducing Prefix C. To Ser.

History $h$: reading → writing

History $h'$: reading → writing

- History $h'$ obtained by splitting each transaction $R_1...R_n$ $W_1...W_m$ of $h$ into two transactions $R_1...R_n$ and $W_1...W_m$

- $h$ satisfies Prefix Consistency $\iff h'$ satisfies Serializability
**Reducing Prefix C. To Ser.**

<table>
<thead>
<tr>
<th>Long fork: (PC violation)</th>
<th>read(x,0)</th>
<th>read(y,0)</th>
<th>read(x,1)</th>
<th>read(y,0)</th>
<th>read(x,0)</th>
<th>read(y,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Long fork transformed: (SER violation)

<table>
<thead>
<tr>
<th>read(x,0)</th>
<th>read(y,0)</th>
<th>read(x,1)</th>
<th>read(y,0)</th>
<th>read(x,0)</th>
<th>read(y,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Long fork:**

- read(x,0)
- read(y,0)
- read(x,1)
- read(y,0)
- read(x,0)
- read(y,1)

**Reduced Prefix:**

- read(x,0)
- read(y,0)
- read(x,1)
- read(y,0)
- read(x,0)
- read(y,1)

**Reduced:**

- x = 1
- y = 1

---

**PC violation:**

- read(x,0)
- read(y,0)
- read(x,1)
- read(y,0)
- read(x,0)
- read(y,1)

**SER violation:**

- read(x,0)
- read(y,0)
- read(x,1)
- read(y,0)
- read(x,0)
- read(y,1)
Reducing Prefix C. To Ser.

<table>
<thead>
<tr>
<th>Long fork: (PC violation)</th>
<th>read(x,0)</th>
<th>read(y,0)</th>
<th>read(x,1)</th>
<th>read(x,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1</td>
<td>y = 1</td>
<td>read(y,0)</td>
<td>read(y,0)</td>
<td>read(y,1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Long fork transformed: (SER violation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x,0)</td>
</tr>
<tr>
<td>x = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lost update: (PC valid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x,0)</td>
</tr>
<tr>
<td>x = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lost update transformed: (SER valid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x,0)</td>
</tr>
<tr>
<td>so</td>
</tr>
<tr>
<td>x = 1</td>
</tr>
</tbody>
</table>

| read(y,0) |                     |
| so        |                     |
| x = 2     |                     |
Reducing Snapshot Isolation To Serializability

- a refinement of the Prefix Consistency reduction
- for pairs of transactions that write to a common variable, we enforce that both “read transactions” cannot be committed before both “write transactions”
- initial history satisfies Snapshot Isolation <=> transformed history satisfies Serializability
Conclusions

Replicated data types are a solution for high-frequency parallel accesses to high-quantity data.

Different trade-offs between consistency and availability.

Reasoning about and on top of weak consistency is challenging.